

Pour Noël, offrez vous le tout dernier logiciel de gestion de rendez vous.

« Avec **OpenAgenda**, gérez votre agenda comme vous l'entendez, sans aucune limite ! »

Mon calendrier

Fichier Editer

Année : 2 013

Semaine : 2

Mois de Janvier

	Lundi 7	Mardi 8	Mercredi 9	Jeudi 10	Vendredi 11	Samedi 12	Dimanche 13
0 h							
1 h							
2 h							
3 h							
4 h							
5 h							
6 h							
7 h							
8 h	Projet	Prog. lin.	Merise	Java TP	BDD TP	BDD TP	
9 h	Projet	Prog. lin.	Merise	Java TP	BDD TP	BDD TP	
10 h	Projet	Expression	Windev	XML TP	Gestion TP		
11 h	Projet	Expression	Windev	XML TP	Gestion TP		
12 h							
13 h							
14 h	Anglais	Java	Gestion	Java DS	Prog. lin.		
15 h	Anglais	Java	Gestion	Java DS	Prog. lin.		
16 h	XML						
17 h	XML						
18 h		BDD	Merise TD				
19 h		BDD	Merise TD				
20 h							
21 h							
22 h							
23 h							

53 µ							
55 µ							
57 µ							
59 µ							
78 µ		BDD	Merise TD				
78 µ		BDD	Merise TD				

Voici tout ce que vous pouvez faire !

- ★ Créez autant d'agendas différents que vous voulez ! Un chacun : pour vous, pour chaque membre de votre famille et pour tous vos amis... Oui, oui, c'est possible !
- ★ En plus, les agendas sont dans un format ouvert et exportable ! Réutilisez les dans toutes vos applications *OpenSuite* !
- ★ Profitez des dernières technologies pour la gestion personnalisée des rendez-vous ! Grâce à la colorisation, chaque rendez-vous peut avoir une couleur différente ! Choisissez votre préférée parmi les 16 millions possibles !
- ★ Placez vos rendez-vous n'importe où dans le temps ! Vous avez rendez-vous avec vos anciens camarades de lycée dans 10 ans ? Pas de souci, *OpenAgenda* est capable de gérer des rendez-vous très éloignés temporellement !
- ★ Il le gère tellement bien que vous pouvez même l'utiliser pour mémoriser des moments passés importants ! Et sait-on jamais, si on revenait dans le temps, vous pourriez assister à un célèbre rendez-vous ! (psst, mettez aussi les numéros du loto!)
- ★ Toujours pratique, votre saisie est assistée ! Plus de tracas pour écrire une date ou un horaire !
- ★ *OpenAgenda* ne serait pas totalement utile si vous ne pouviez pas ajouter une description à un rendez-vous, histoire de vous rappeler de certains trucs... Heureusement, *OpenAgenda* vous le permet !
- ★ Vous êtes pressé ? Pas de problèmes, fermez l'agenda et *OpenAgenda* vous le sauve tout seul comme un grand !
- ★ Encore plus terrible, *OpenAgenda* vous réouvre le dernier agenda, toujours tout seul, au prochain lancement !
- ★ Oups, un mauvais clic ou une mauvaise saisie ? Arrêtez de pleurer, faites CTRL+Z !
- ★ Encore raté ? Faites le encore !
- ★ Trop d'annulation ? Passez au CTRL+Y !
- ★ Vous n'êtes pas à l'aise avec les dates ? Utilisez l'assistant graphique de création ! Quoi, vous ne savez pas ce que c'est ?! Ce gadget indispensable vous permet tout bonnement de cliquer-glisser pour créer votre rendez-vous ! Oh, comme par magie les dates correspondent déjà !
- ★ Vous vous êtes bien amusés mais vous avez fait n'importe quoi ? Double clic gauche pour éditer et double clic droit pour supprimer !
- ★ Hey, vous pouvez annuler une suppression ! C'est *OpenAgenda*, pas ArgoUML !
- ★ Profitez-en ! C'est Noël !

Rapport TP 6 – Agenda

Le but du TP est de montrer que nous sommes maintenant capables de développer une vraie application Java, comme des grands. Cette application est concrètement un gestionnaire de rendez-vous, présentés sous la forme d'un agenda.

Le projet contient 3 packages et 22 classes.

L'application est décomposée selon le patron d'architecture MVC

➤ **Modèle**

- **RendezVous**, représentant une entité rendez-vous ;
- **RendezVousComparator**, pour comparer deux rendez-vous ;
- **RendezVousFactory**, pour sauver ou charger depuis un fichier ;
- **AgendaTableModel**, pour garder les rendez-vous filtrés ;
- **AgendaTableColumnModel**, pour garder les en-têtes.

➤ **Vue**

- **TimeMachinePane**, panneau personnalisé pour changer de semaine ;
- **RendezVousDialog**, pour la création ou l'édition d'un rendez-vous ;
- **RendezVousDialogArg**, pour récupérer l'édition de l'utilisateur ;
- **AgendaTableCellRenderer**, moteur de rendu abstrait d'une cellule ;
- **AgendaHourRenderer**, moteur de rendu des heures ;
- **AgendaRendezVousRenderer**, moteur de rendu d'un rendez-vous ;
- **MouseMotion**, pour gérer les déplacements de la souris ;
- **MainFrame**, notre fenêtre principale.

➤ **Contrôleur**

- **Utils**, classe proposant des outils ;
- **TimeMachineController**, pour gérer la période visualisée ;
- **Command**, fonctionnalité logicielle abstraite ;
- **CommandAddRendezVous**, action associée à l'ajout d'un rendez-vous ;
- **CommandModifyRendezVous**, action associée à la modification ;
- **CommandRemoveRendezVous**, action associée à la suppression ;
- **UndoRedoManager**, gestionnaire de l'exécution ou de l'annulation des actions ;
- **Agenda**, gère les rendez-vous et le filtrage en fonction de la période visualisée ;
- **Main**, classe principale, point d'entrée du programme.

Dans ce rapport sera donné le strict minimum de code, sinon il aurait fallu une vingtaine de pages supplémentaires uniquement remplies de code... Il est donc conseillé de lire les sources !

Sommaire

Modèle → RendezVous.....	5
Modèle → RendezVousComparator.....	7
Modèle → RendezVousFactory.....	8
Modèle → AgendaTableModel.....	10
Modèle → AgendaTableColumnModel.....	12
Vue → TimeMachinePane.....	13
Vue → RendezVousDialog ; Vue → RendezVousDialogArg.....	14
Vue → AgendaTableCellRenderer.....	17
Vue → AgendaHourRenderer ; Vue → AgendaRendezVousRenderer.....	18
Vue → MouseMotion.....	19
Contrôleur → Utils.....	20
Contrôleur → TimeMachineController.....	21
Contrôleur → Command.....	23
Contrôleur → CommandAddRendezVous.....	24
Contrôleur → CommandModifyRendezVous.....	24
Contrôleur → CommandRemoveRendezVous.....	25
Contrôleur → UndoRedoManager.....	26
Contrôleur → Agenda.....	27
Vue → MainFrame.....	28
Contrôleur → Main.....	31

Modèle → RendezVous

Un rendez-vous est représenté par :

- Un titre, de type **String** ;
- Une date et heure de début, de type **Date** ;
- Une date et heure de fin, de type **Date** ;
- Une description, de type **String** ;
- Une couleur, de type **Color**.

La classe définissant un rendez-vous propose naturellement les accesseurs en lecture et écriture. Elle propose aussi un setter global qui permet de recopier toutes les valeurs d'un autre rendez-vous en bloc :

```
public void setByCopy(RendezVous other) {  
    setTitle(other.getTitle());  
    setBegin(other.getBegin());  
    setEnd(other.getEnd());  
    setSummary(other.getSummary());  
    setColor(other.getColor());  
}
```

La classe propose de nombreux constructeurs utiles comme un constructeur par copie et particulièrement un constructeur par défaut afin de respecter la convention JavaBean. Ainsi l'exportation ou l'importation d'un rendez-vous sera facilitée.

L'égalité de deux rendez-vous est basée sur l'égalité de leurs dates de début, dates de fin et titres.

Modèle → RendezVousComparator

Il est plus intéressant de définir un critère de comparaison externe à la classe **RendezVous**, ainsi on peut trier ces derniers selon divers ordres (par date de début, par date de fin, etc...). Cette classe implémente **Comparator<RendezVous>** et est évidemment un singleton, puisque la définition d'un ordre est propre à la classe.

Dans l'application actuelle, l'algorithme de filtre n'est pas basé sur le tri des rendez-vous mais en algorithme futur pourrait être amené à utiliser différents ordres de tri.

Cette classe n'est actuellement utilisée que lors du chargement et de la sauvegarde de l'agenda où l'on effectue un tri afin un fichier temporellement cohérent.

L'ordre choisi dans cette implémentation est celui de classer les rendez-vous par date de début, et en cas d'égalité, par date de fin.

```
/**
 * Comparer deux rendez vous.
 * Ordre : date de début, puis date de fin
 * @param rdv1 Premier rendez vous
 * @param rdv2 Second rendez vous
 * @return Différence de position entre les deux rendez vous
 */
@Override
public int compare(RendezVous rdv1, RendezVous rdv2) {
    // divise pour avoir une précision à la minute seulement
    int r = (int)((rdv1.getBegin().getTime() - rdv2.getBegin().getTime()) / (1000*60));
    if (r == 0)
        r = (int)((rdv1.getEnd().getTime() - rdv2.getEnd().getTime()) / (1000*60));
    return r;
}
```

Modèle → RendezVousFactory

Cette classe, étant un singleton, permet de sauvegarder les rendez-vous dans un fichier et de charger les rendez-vous d'un fichier.

Étant donné que notre classe **RendezVous** est un JavaBean, on utilise les classes de la bibliothèque standard : **java.beans.XMLEncoder** et **java.beans.XMLDecoder**.

Cette dans cette classe que l'on alloue le type réel de l'ensemble des rendez-vous.

```
/**
 * Obtenir un ensemble vide de rendez vous
 * @return Ensemble vide de rendez vous
 */
public List<RendezVous> getEmptySet() {
    return new LinkedList<RendezVous>();
}
```

```
/**
 * Sauvegarder un agenda dans un fichier
 * @param setRendezVous Ensemble de rendez vous
 * @param destination Fichier de destination
 */
public void save(List<RendezVous> setRendezVous, File destination) {
    Collections.sort(setRendezVous, RendezVousComparator.getInstance());
    try {
        XMLEncoder e = new XMLEncoder(new BufferedOutputStream(
            new FileOutputStream(destination)));
        for (RendezVous rendezVous : setRendezVous) {
            e.writeObject(rendezVous);
        }
        e.close();
    } catch (FileNotFoundException e) {
        System.err.println("When saving agenda : " + e.getMessage());
    }
}
```

```
/**
 * Charger un agenda depuis un fichier
 * @param source Fichier source
 * @return Ensemble des rendez vous
 */
public List<RendezVous> load(File source) {
    List<RendezVous> set = getEmptySet();
    XMLDecoder xmlDecoder = null;
    try {
        xmlDecoder = new XMLDecoder(new BufferedInputStream(new FileInputStream(source)));
        Object readObj;
        while ((readObj = xmlDecoder.readObject()) != null) {
            try {

```

```
        set.add((RendezVous) readObj);
    } catch (ClassCastException e) {
        System.err.println("When loading agenda : " + e.getMessage());
    }
}
} catch (IndexOutOfBoundsException e) { // see XMLDecoder.readObject() documentation
} catch (FileNotFoundException e) {
    System.err.println("When loading agenda : " + e.getMessage());
} finally {
    if (xmlDecoder != null)
        xmlDecoder.close();
}
Collections.sort(set, RendezVousComparator.getInstance());
return set;
}
```


Modèle → AgendaTableModel

La **JTable** permettant d'afficher les rendez-vous d'une certaine semaine a besoin de réaliser de nombreux accès en lecture aux données des rendez-vous. Pour éviter de filtrer tous les rendez-vous à chaque requête, on utilise un modèle de table qui sera mis à jour seulement lorsque l'utilisateur change de période visualisée.

Notre classe hérite donc de **DefaultTableModel** pour profiter du conteneur déjà en place dans cette classe. Le conteneur possède concrètement 24 lignes pour représenter les 24 heures d'une journée standard et 8 colonnes, une pour les heures (0h, 1h, ..., 23h) et les sept autres pour les jours de la semaine, du lundi au dimanche.

Lorsque ce conteneur doit être mis à jour, on efface les anciennes données, puis pour chaque rendez-vous, on vérifie que ses dates sont comprises dans la période visualisée. On prend bien sûr en compte le cas où un rendez-vous serait à cheval entre deux semaines.

```
/**
 * Mettre à jour la mémoire tampon avec les rendez-vous filtrés.
 * @param begin Date de début de la période de filtrage
 * @param end Date de fin de la période de filtrage
 * @param mCollection Ensemble des rendez vous à filtrer
 */
public void updateData(Date begin, Date end, List<RendezVous> mCollection) {
    clear();
    for (RendezVous rendezVous : mCollection)
        if (rendezVous.getBegin().before(end) || rendezVous.getEnd().after(begin))
            printRendezVous(rendezVous, begin);
}
```

Ensuite, on calcule la position de la cellule où débutera le rendez-vous, relativement au début de la période visualisée. De même pour la case d'arrivée. Puis, case par case, nous inscrivons les données dans le tableau.

```
/**
 * Ecrire les informations d'un rendez vous dans les bonnes cellules
 * @param rdv Rendez-vous à traiter
 * @param axe Date du début de la période visualisée
 */
private void printRendezVous(RendezVous rdv, Date axe) {
    int caseDepart = calculatePositionCell(rdv.getBegin(), axe);
    int caseArrivee = calculatePositionCell(rdv.getEnd(), axe);

    while (caseDepart < caseArrivee) {
        int colonne = caseDepart / NB_HOURS;
        int ligne = caseDepart - NB_HOURS*colonne;
        ++colonne;

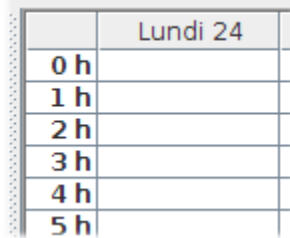
        if (colonne > 0 && colonne < NB_COLS && ligne >= 0 && ligne < NB_HOURS)
            setValueAt(rdv, ligne, colonne);
        ++caseDepart;
    }
}
```

Moins technique, ce modèle indique le type des données contenues dans les colonnes.

```
/**
 * Obtenir le type des données d'une colonne
 * @param columnIndex Index de la colonne
 * @return Métaclasse d'une possible donnée
 */
@Override
public Class<?> getColumnClass(int columnIndex) {
    return (columnIndex > 0) ? RendezVous.class : String.class;
}
```

Modèle → AgendaTableColumnModel

Dans la continuité de la personnalisation de notre **JTable**, nous utilisons un modèle de colonnes qui nous permettra de gérer les en-têtes et d'afficher les jours et leur date dans les titres des colonnes. Cette classe hérite de **DefaultTableColumnModel**. Seule la première colonne est spéciale puisqu'elle contient les heures. On lui fixe donc une petite largeur pour question d'esthétique.



	Lundi 24
0 h	
1 h	
2 h	
3 h	
4 h	
5 h	

Lorsque la période visualisée change, ce modèle en est notifié et peut donc mettre à jour les en-têtes.

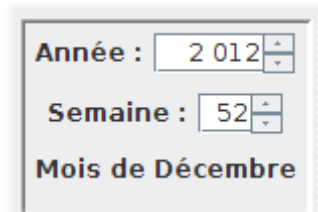
```
/**
 * Mettre à jour les en-têtes des colonnes
 * @param begin Date du début de la semaine
 */
public void updateHeader(Date begin) {
    mCal.setTime(begin);
    for (int i=1; i < AgendaTableModel.NB_COLS; ++i) {
        getColumn(i).setHeaderValue(formatHeader(mCal));
        fireChange(i);
        mCal.add(Calendar.DATE, 1); // increment
    }
}
```

Le formatage des en-têtes correspond à la concaténation du jour de la semaine et son numéro dans le mois.

```
/**
 * Obtenir un texte mis en forme de la date
 * @param cal Calendrier contenant la date
 * @return Texte représentant la date (ex:Lundi 11)
 */
private String formatHeader(Calendar cal) {
    return String.format("%s %s",
        Utils.capitalize(cal.getDisplayName(Calendar.DAY_OF_WEEK,
            Calendar.LONG, Locale.getDefault())),
        cal.get(Calendar.DAY_OF_MONTH));
}
```

Vue → TimeMachinePane

Réaliser un panneau personnalisé est conceptuellement intéressant, car cela permet de regrouper un ensemble de composants thématiquement proches. Notre classe étend donc un **JPanel**.



Le nouveau composant introduit dans ce TP est le **JSpinner**. Il est très utile pour aider à la saisie. On peut lui associer un modèle indiquant, entre autres, les valeurs maximales et minimales que peut contenir le champ, ce qui épargne bien des soucis aux développeurs.

On utilise ce modèle pour le spinner de la semaine. En effet, le nombre de semaines maximales dépend de l'année. Une définition simple pour trouver le nombre maximal de semaines est la suivante : « La dernière semaine de l'année ISO est celle du 28 décembre ».

```
/** Obtenir un modèle de spinner pour la semaine
 * @param year Année en cours
 * @return Un modèle dont les min et max sont adaptés
 */
private SpinnerNumberModel getWeekSpinnerModel(int week) {
    Calendar cal = mController.getCal();
    cal.set(cal.get(Calendar.YEAR), Calendar.DECEMBER, 28);

    int min = cal.getMinimum(Calendar.WEEK_OF_YEAR);
    int max = cal.get(Calendar.WEEK_OF_YEAR);
    week = (week < min) ? min : week;
    week = (week > max) ? max : week;
    return new SpinnerNumberModel((Number) week, min, max, 1);
}
```

Cette classe n'étant responsable que de l'interaction avec l'utilisateur, elle notifie un contrôleur (de type **TimeMachineController**) lorsque les valeurs ont changées.

```
/**
 * Méthode appelée lorsque que les valeurs des spinner ont changées
 * @param event Argument d'événement
 */
@Override
public void stateChanged(ChangeEvent event) {
    if (event.getSource() == mSpYear || event.getSource() == mSpWeek) {
        mController.setView((int)mSpYear.getValue(), (int)mSpWeek.getValue());
        if (event.getSource() == mSpYear) // met à jour les valeurs possibles des semaines
            mSpWeek.setModel(getWeekSpinnerModel((int)mSpWeek.getValue()));
        mLblMonth.setText(getLabelMonth());
    }
}
```

Vue → **RendezVousDialog**; **Vue** → **RendezVousDialogArg**

Notre application étant un gestionnaire de rendez-vous, il faut bien un moyen à l'utilisateur de renseigner les champs décrivant un nouveau rendez-vous. Cette classe propose donc, au travers d'une boîte de dialogue personnalisée, un formulaire répondant à ce besoin. Cette classe hérite donc de **JDialog**.

Afin d'en faire une classe réutilisable, elle est conçue pour éditer un rendez-vous, qu'il soit nouveau ou non. Elle permet aussi d'annuler les modifications sans que le rendez-vous édité ne soit affecté. Pour cela, le développeur communique avec cette boîte de dialogue au travers d'un objet de type **RendezVousDialogArg**.

```
/**
 * Rendez vous à éditer
 */
private RendezVous mRendezVous;
/**
 * Indique si l'utilisateur a validé les modifications
 */
private boolean mSubmitted;
/**
 * Constructeur
 * @param rendezVous Rendez vous à éditer
 */
public RendezVousDialogArg(RendezVous rendezVous) {
    this.mRendezVous = rendezVous;
    this.mSubmitted = false;
}
```

Ainsi, sachant que le formulaire est modal, l'édition d'un rendez-vous se fait en une ligne de code :

```
new RendezVousDialog(frame, new RendezVousDialogArg(rendezVous));
```

Rendez-vous

Titre :

Date début (JJ/MM/AAAA) :

Heure de début (HH:MM) :

Date fin (JJ/MM/AAAA) :

Heure de fin (HH:MM) :

Description (facultative) :

Couleur d'affichage :

La saisie d'une date ou d'un horaire pouvant être compliquée, sa vérification l'est encore plus ! Heureusement, Java propose des composants texte qui n'accepte qu'une saisie formatée, ce sont les **JFormattedTextField**.

```
// Patterns à respecter dans les champs de date et d'horaire
SimpleDateFormat dayFormat = new SimpleDateFormat("dd/MM/yyyy");
SimpleDateFormat timeFormat = new SimpleDateFormat("HH:mm");

mTfBeginDay = new JFormattedTextField(dayFormat);
mTfBeginDay.setValue(mRendezVousSource.getBegin());
mTfBeginDay.addActionListener(this);

mTfBeginTime = new JFormattedTextField(timeFormat);
mTfBeginTime.setValue(mRendezVousSource.getBegin());
mTfBeginTime.addActionListener(this);
```

Le placement des composants est un peu compliqué. L'utilisation manuelle d'un Layout complexe semblant peu judicieuse, nous avons donc séparé en panneaux les champs de textes et leurs labels associés. Nous avons fixé une taille unique pour les champs et indiqué au FlowLayout d'aligner les composants à droite. Finalement, on ajoute tous ces panneaux :

```
JPanel pTitle = new JPanel(new FlowLayout(FlowLayout.RIGHT, 5, 1));
pTitle.add(mLblTitle); pTitle.add(mTfTitle);
// [...]
JPanel contentPane = new JPanel();
contentPane.setLayout(new BoxLayout(contentPane, BoxLayout.Y_AXIS));
contentPane.add(pTitle);      contentPane.add(pBeginDay);
contentPane.add(pBeginTime); contentPane.add(pEndDay);
contentPane.add(pEndTime);   contentPane.add(pSummary);
contentPane.add(pColor);     contentPane.add(pButtons);

contentPane.setBorder(BorderFactory.createEmptyBorder(10, 5, 10, 5));
this.setContentPane(contentPane);
```

Java étant un langage moderne, sa bibliothèque est bien remplie ! Il est donc aussi aisé de faire choisir une couleur à un utilisateur que de lui faire sélectionner un fichier.

```
@Override
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == mBtColor) {
        Color newColor = JColorChooser.showDialog(this,
            "Choisissez votre couleur préférée", mRendezVousColor);
        if (newColor != null) {
            mRendezVousColor = newColor;
            mBtColor.setBackground(mRendezVousColor);
        }
    }
}
```

Lorsque l'utilisateur annule les modifications, la conception de la boîte de dialogue permet de faire uniquement ceci :

```
else if (e.getSource() == mBtCancel) {
    dispose();
}
```

La validation entraîne quelques vérifications : par exemple les cinq premiers champs doivent être remplis et la date de fin doit être postérieure à celle du début.

Afin de récupérer une seule date fusionnant les deux informations disponibles, soit la date du jour et l'horaire, on réalise les quelques instructions ci-dessous :

```
/** Obtenir une date complète (jour + horaire)
 * à partir de deux champs de texte formatés
 * @param tfDay Champ contenant la date du jour
 * @param tfTime Champ contenant l'horaire
 * @return Une date issue de la fusion des paramètres
 */
private Date getDateFromTextFields(JFormattedTextField tfDay, JFormattedTextField tfTime) {
    Date day = (Date) tfDay.getValue();
    Date time = (Date) tfTime.getValue();

    Calendar cal = GregorianCalendar.getInstance();

    // obtenir les infos du jour
    cal.setTime(day);
    int year = cal.get(Calendar.YEAR);
    int month = cal.get(Calendar.MONTH);
    int dayMonth = cal.get(Calendar.DAY_OF_MONTH);

    // obtenir les infos de l'horaire
    cal.setTime(time);
    int hour = cal.get(Calendar.HOUR_OF_DAY);
    int mins = cal.get(Calendar.MINUTE);

    // fusionner les infos
    cal.set(year, month, dayMonth, hour, mins);
    return cal.getTime();
}
```

Une fois que tout est validé, on modifie les attributs du rendez-vous source.

```
// mise à jour de l'instance source
mRendezVousArg.setSubmitted();
mRendezVousSource.setTitle(title);
mRendezVousSource.setBegin(begin);
mRendezVousSource.setEnd(end);
mRendezVousSource.setSummary(mTfSummary.getText());
mRendezVousSource.setColor(mRendezVousColor);
dispose();
```

Vue → AgendaTableCellRenderer

Cette classe est abstraite, elle permet d'avoir les bases d'un moteur de rendu d'une cellule de table. Elle étend donc **DefaultTableCellRenderer**. La méthode d'affichage adoptée consiste à n'utiliser qu'un seul **JPanel** et un seul **JLabel** contenu dans ce dernier.

```
/**
 * Panneau renvoyé pour l'affichage
 */
protected JPanel mPanel;
/**
 * Label contenant le texte à afficher
 */
protected JLabel mLabel;
/**
 * Constructeur
 */
public AgendaTableCellRenderer() {
    mPanel = new JPanel();
    mPanel.setLayout(new GridBagLayout());
    mLabel = new JLabel();
    mPanel.add(mLabel);
}
```

Il suffira aux classes filles de changer le contenu du label et éventuellement la couleur de fond du panneau puis de retourner ce dernier pour être utilisé pour afficher une cellule. Puisque la **JTable** n'utilise le panneau que comme modèle graphique, l'unicité du panneau n'engendre pas de problème.

Vue → **AgendaHourRenderer** ; **Vue** → **AgendaRendezVousRenderer**

La première classe, fille de la précédente, permet d'afficher les heures dans la première colonne.

```
public class AgendaHourRenderer extends AgendaTableCellRenderer {

    public AgendaHourRenderer() {
        super();
        mPanel.setLayout(new FlowLayout(FlowLayout.RIGHT,2,0));
        mPanel.setBackground(Color.WHITE);
    }

    @Override
    public Component getTableCellRendererComponent(JTable table, Object value,
        boolean isSelected, boolean hasFocus, int row, int column) {

        mLabel.setText(value.toString());
        return mPanel;
    }
}
```

Le moteur de rendu des rendez-vous n'est pas plus complexe que le précédent. Il hérite aussi de **AgendaTableCellRenderer**.

Si la cellule ne contient pas de rendez-vous, pas besoin de continuer le traitement. Sinon, le label contient le titre du rendez-vous et on applique au panneau la couleur d'affichage du rendez-vous.

```
@Override
public Component getTableCellRendererComponent(JTable table, Object value,
    boolean isSelected, boolean hasFocus, int row, int column) {

    if (value == null)
        return null;

    RendezVous rendezVous = (RendezVous) value;
    mLabel.setText(rendezVous.getTitle());
    mPanel.setBackground(rendezVous.getColor());

    return mPanel;
}
```

Vue → MouseMotion

La gestion du déplacement de la souris est utile pour la création de rendez-vous graphiquement. L'utilisateur clique sur la cellule où commence son rendez-vous, glisse la souris jusqu'à la cellule où il se termine et relâche. Pour que l'utilisateur puisse aussi commencer par la fin du rendez-vous et terminer par le début, il faut utiliser cette classe qui assure l'ordre des coordonnées. La première coordonnée correspondra, graphiquement, toujours à la date la plus ancienne.

```
/**
 * Spécifier les coordonnées de départ
 * @param row Ligne de départ
 * @param col Colonne de départ
 */
public void setFirstRowCol(int row, int col) {
    row1 = row;
    col1 = col;
    mValid = false;
}
```

```
/**
 * Spécifier les coordonnées d'arrivée
 * @param row Ligne d'arrivée
 * @param col Colonne d'arrivée
 */
public void setSecondRowCol(int row, int col) {
    if (col < col1 || (col == col1 && row < row1)) {
        // permutation
        col2 = col1;
        col1 = col;
        row2 = row1;
        row1 = row;
    } else {
        row2 = row;
        col2 = col;
    }
    mValid = true;
}
```

Contrôleur → Utils

Cette petite classe propose un endroit où mettre des méthodes utiles et communes à l'ensemble de l'application. Pour des questions de performances, il est préférable d'en faire un singleton que d'écrire des méthodes de classes.

Actuellement, il n'y a qu'une méthode pour manipuler une chaîne.

```
/**
 * Mettre la première lettre en majuscule
 * @param input Texte à traiter
 * @return Texte dont la première lettre est en majuscule
 */
public String capitalize(String input) {
    if (input != null && input.length() > 0)
        input = input.substring(0, 1).toUpperCase() + input.substring(1);
    return input;
}
```

Contrôleur → TimeMachineController

Pour gérer concrètement la période que l'utilisateur est en train de visualiser, il nous faut un objet qui sera capable de transformer une cellule en date et vice-versa. Un objet qui sera capable de mettre à jour ce qu'il faut et quand il le faut ! Cet objet, c'est le **TimeMachineController** !

Cette classe gérant directement du temps, elle est très dépendante de **java.util.Calendar**. Elle est ainsi capable se mettre correctement à jour lorsque l'utilisateur change d'année ou de semaine dans le **TimeMachinePane**.

```
/**
 * Année et semaine visualisés
 */
private int mYear, mWeek;
/**
 * Calendrier utile aux calculs
 */
private Calendar mCal;
/**
 * Date de début de la période visualisée
 */
private Date mFirstDate;
/**
 * Date de fin (exclusive) de la période visualisée
 */
private Date mEndDate;
```

Lorsque l'utilisateur change de période visualisée, un appel à la méthode suivante est effectué. Elle se charge de mettre à jour le contenu de l'objet typé **AgendaTableModel**.

```
/**
 * Définir l'année et la semaine de la période visualisée
 * @param year Année
 * @param week Semaine
 */
public void setView(int year, int week) {
    mYear = year;
    mWeek = week;
    upDates();

    Agenda.getInstance().setFirstDate(getFirstDate(), getEndDate());
}
```

```
/**
 * Recalculer les dates de début et de fin
 */
private void upDates() {
    mCal.clear();
```

```
mCal.set(Calendar.YEAR, mYear);
mCal.set(Calendar.WEEK_OF_YEAR, mWeek);
mCal.set(Calendar.DAY_OF_WEEK, Calendar.MONDAY);
mFirstDate = mCal.getTime();

mCal.set(Calendar.DAY_OF_WEEK, Calendar.SUNDAY);
mCal.add(Calendar.HOUR_OF_DAY, 24); // end is actually the sunday's end
mEndDate = mCal.getTime();
}
```

Afin de convertir la position d'une cellule en date, on utilise la méthode suivante. Cette méthode n'est pas liée au concept de tableau mais cela fonctionne puisque, par convention, une colonne représente un jour et une ligne une heure.

```
/**
 * Obtenir la date après tant de jours et d'heures
 * après le début de la période visualisée
 * @param nbDays Nombre de jours
 * @param nbHours Nombre d'heures
 * @return
 */
public Date getAfter(int nbDays, int nbHours) {
    mCal.setTime(getFirstDate());
    mCal.add(Calendar.DAY_OF_MONTH, nbDays);
    mCal.add(Calendar.HOUR_OF_DAY, nbHours);
    return mCal.getTime();
}
```

Contrôleur → Command

Cette classe abstraite est la base du patron de conception portant le même nom. Elle sert à abstraire une action utilisateur et permet de très facilement l'annuler.

```
public abstract class Command {  
    /**  
     * Exécuter la commande  
     */  
    public abstract void execute();  
    /**  
     * Annuler les effets de l'exécution  
     */  
    public abstract void undo();  
}
```

Contrôleur → CommandAddRendezVous

Représente la création d'un nouveau rendez-vous. On mémorise le rendez-vous en question afin de pouvoir le supprimer ou le réajouter.

```
public class CommandAddRendezVous extends Command {
    /** Rendez vous créé */
    private RendezVous mRendezVous;
    /**
     * Constructeur
     * @param rendezVous Nouveau rendez vous à ajouter
     */
    public CommandAddRendezVous(RendezVous rendezVous) {
        mRendezVous = new RendezVous(rendezVous);
    }
    @Override
    public void execute() {
        Agenda.getInstance().add(mRendezVous);
    }
    @Override
    public void undo() {
        Agenda.getInstance().remove(mRendezVous);
    }
}
```

Contrôleur → CommandModifyRendezVous

Représente la modification d'un rendez-vous. On mémorise une référence sur le rendez-vous concerné ainsi que ses anciens et nouveaux attributs.

```
public class CommandModifyRendezVous extends Command {
    /** Instance du rendez vous modifié */
    private RendezVous mRendezVousSource;
    /** Anciens attributs du rendez vous modifié */
    private RendezVous mOldAttributes;
    /** Nouveaux attributs du rendez vous modifié */
    private RendezVous mNewAttributes;

    /** Constructeur
     * @param rendezVous Rendez vous à modifier
     * @param newAttributes Nouveaux attributs
     */
    public CommandModifyRendezVous(RendezVous rendezVous, RendezVous newAttributes) {
        mRendezVousSource = rendezVous;
        mNewAttributes = new RendezVous(newAttributes);
        mOldAttributes = new RendezVous(rendezVous);
    }
}
```

```
/** Exécuter les modifications du rendez vous */
@Override
public void execute() {
    mRendezVousSource.setByCopy(mNewAttributes);
    // indique que le rendez vous a été modifié
    Agenda.getInstance().setDirty(mRendezVousSource);
}
/** Annule les modifications */
@Override
public void undo() {
    mRendezVousSource.setByCopy(mOldAttributes);
    Agenda.getInstance().setDirty(mRendezVousSource);
}
}
```

Contrôleur → CommandRemoveRendezVous

Représente la suppression d'un rendez-vous. On mémorise le rendez-vous en question afin de pouvoir le supprimer ou le réajouter.

```
public class CommandRemoveRendezVous extends Command {
    /** Rendez vous supprimé */
    private RendezVous mRendezVous;
    /** Constructeur
     * @param rendezVous Rendez vous à supprimer
     */
    public CommandRemoveRendezVous(RendezVous rendezVous) {
        mRendezVous = rendezVous;
    }
    /** Exécute la suppression du rendez vous */
    @Override
    public void execute() {
        Agenda.getInstance().remove(mRendezVous);
    }
    /** Annule la suppression, soit réajoute le rendez vous */
    @Override
    public void undo() {
        Agenda.getInstance().add(mRendezVous);
    }
}
```


Contrôleur → UndoRedoManager

Toujours dans la suite de l'implémentation du patron de conception commande, nous avons développé une classe utilitaire. Elle est chargée de maintenir deux piles : la première contenant les commandes effectuées et la seconde les commandes annulées.

En plus de cela, elle active ou désactive les composants associés à l'annulation et à la réexécution.

```
/**
 * Constructor
 * @param undo Component linked with undo action
 * @param redo Component linked with redo action
 */
public UndoRedoManager(JComponent undo, JComponent redo) {
    mCommandsDone = new Stack<>();
    mCommandsUndone = new Stack<>();
    mUndo = undo;
    mRedo = redo;
    updateMenuItems();
}
```

Lorsque l'utilisateur effectue une action, on crée la commande adéquate que l'on passe au gestionnaire.

```
/** Execute the command and remember it
 * @param com Command to execute
 */
public void execute(Command com) {
    com.execute();
    mCommandsDone.push(com);
    mCommandsUndone.clear(); // clear the 'branch'
    updateMenuItems();
}
```

Puis si l'utilisateur souhaite annuler ou refaire son action, on l'indique au gestionnaire.

```
/** Cancel the latest executed command */
public void undo() {
    Command com = mCommandsDone.pop();
    com.undo();
    mCommandsUndone.push(com);
    updateMenuItems();
}

/** Re-execute the latest canceled command */
public void redo() {
    Command com = mCommandsUndone.pop();
    com.execute();
    mCommandsDone.push(com);
    updateMenuItems();
}
```

Contrôleur → Agenda

Comme la fenêtre principale est la charnière des éléments, l'**Agenda** est le point central des interactions avec le modèle. Pour faciliter le développement, cette classe est un singleton.

Cette classe gère l'ensemble des rendez-vous et propose donc d'ajouter un rendez-vous, de supprimer un ou tous les rendez-vous, d'obtenir leur nombre, de sauvegarder l'ensemble dans un fichier et de charger un ensemble de rendez-vous depuis un fichier.

Cette classe propose aussi les modèles utiles à la **JTable**. Ainsi, lorsqu'un changement de période visualisée est notifiée, tous les éléments sont mis à jour.

```
/**
 * Fixer les dates de la période à filter
 * @param firstDate Début de la période
 * @param endDate Fin (exclusive) de la période
 */
public void setFirstDate(Date firstDate, Date endDate) {
    // se souvenir des critères pour pouvoir updatier
    mFirstDate = firstDate;
    mEndDate = endDate;

    tableColumnModel.updateHeader(mFirstDate);
    update();
}
/**
 * Mettre à jour le tableau d'affichage
 */
private void update() {
    tableModel.updateData(mFirstDate, mEndDate, mSetRendezVous);
}
```

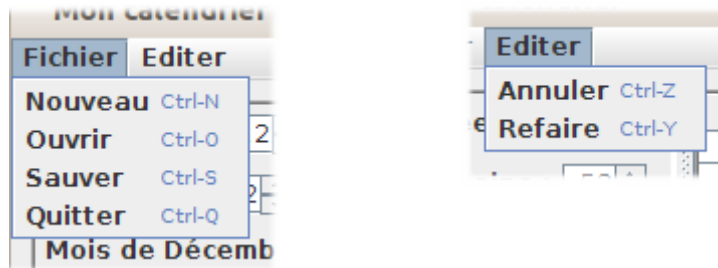
Dans un autre cas, lorsqu'un rendez-vous a été modifié, il peut être utile de rafraîchir ses données en mémoire tampon (**AgendaTableModel**). Pour cela, il suffit de l'indiquer.

```
/**
 * Spécifier qu'un rendez vous a été modifié
 * @param rendezVous Rendez vous mis à jour
 */
public void setDirty(RendezVous rendezVous) {
    if (rendezVous.getBegin().before(mEndDate) || rendezVous.getEnd().after(mFirstDate))
        update();
}
```

Vue → MainFrame

La fenêtre principale contient une barre de menu et un panneau de contenu. Ce dernier est décomposé en deux parties grâce à un **JSplitPane**. À gauche on retrouve un panneau de type **TimeMachinePane** et à droite, une **JTable** affichant un tableau des rendez-vous de la semaine.

La barre de menu contient deux menus, un pour gérer l'agenda, et un autre pour gérer les actions effectuées.



Pour répondre au déclenchement de ces actionneurs, notre fenêtre implémente **ActionListener** et la méthode associée.

```
/** Fermeture de l'application
 */
private void quit() {
    saveCurrentAgenda();
    dispose();
}
@Override
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == mMiNew) {
        saveCurrentAgenda();
        newAgenda();
    } else if (e.getSource() == mMiOpen) {
        saveCurrentAgenda();
        openAgendaFile();
    } else if (e.getSource() == mMiSave) {
        saveCurrentAgenda();
    } else if (e.getSource() == mMiQuit) {
        quit();
    } else if (e.getSource() == mMiUndo) {
        urManager.undo();
    } else if (e.getSource() == mMiRedo) {
        urManager.redo();
    }
    calendrier.repaint();
}
```

L'action associée à l'entrée « Quitter » est la même que celle effectuée lorsque l'utilisateur ferme l'application. On peut traiter cet événement grâce à l'interface **WindowListener**.

```
@Override
public void windowClosing(WindowEvent e) {
    quit();
}
```

La création des composants dans la partie gauche est simplifiée puisqu'elle est déléguée à notre classe **TimeMachinePane**.

```
mTimeMachineController = new TimeMachineController();
mTimeMachinePane = new TimeMachinePane(mTimeMachineController);
```

La partie droite est uniquement constituée du calendrier implémenter par une **JTable**. On personnalise à souhait cette table grâce à toutes les autres développées.

```
mCalendrier = new JTable();
mCalendrier.setRowSelectionAllowed(false);
mCalendrier.getTableHeader().setReorderingAllowed(false);

mCalendrier.setModel(Agenda.getInstance().getTableModel());
mCalendrier.setColumnModel(Agenda.getInstance().getTableColumnModel());
mCalendrier.setDefaultRenderer(RendezVous.class, new AgendaRendezVousRenderer());
mCalendrier.setDefaultRenderer(String.class, new AgendaHourRenderer());
mCalendrier.addMouseListener(this);
```

Toute la partie intéressante réside dans la création de rendez-vous de manière graphique. En effet, il suffit de double cliquer dans une cellule pour créer un rendez-vous dans la cellule ou éditer le rendez-vous précédemment créé. Il suffit aussi de cliquer-glisser pour définir un rendez-vous sur toute une plage horaire.

Ce qu'il faut noter, c'est que la première manière de faire n'est qu'un cas particulier de la deuxième. Dans un premier temps, au clic de la souris, on mémorise les coordonnées de départ de la sélection.

```
@Override
public void mousePressed(MouseEvent e) {
    mouseDrag.setFirstRowCol(mCalendrier.rowAtPoint(e.getPoint()),
        mCalendrier.columnAtPoint(e.getPoint()));
}
```

Ensuite, au relâchement du clic, on mémorise les coordonnées d'arrivée. En fonction d'un éventuel déplacement, du nombre de clic et du bouton de la souris, on effectue l'action associée.

```
@Override
public void mouseReleased(MouseEvent e) {
    mouseDrag.setSecondRowCol(mCalendrier.rowAtPoint(e.getPoint()),
        mCalendrier.columnAtPoint(e.getPoint()));

    if (mouseDrag.isValid()
        && (mouseDrag.hasMoved() || e.getClickCount() == 2)
        && mouseDrag.col1 > 0) {

        // obtenir le rendez vous de la cellule de départ
        RendezVousDialogArg editorArg = new RendezVousDialogArg(
            (RendezVous) Agenda.getInstance().getTableModel()
                .getValueAt(mouseDrag.row1, mouseDrag.col1));

        // s'il est nul, alors il faut un créer un nouveau
        if (editorArg.getRendezVous() == null) {
            editorArg.setRendezVous(new RendezVous(
                mTimeMachineController.getAfter(mouseDrag.col1-1, mouseDrag.row1),
                mTimeMachineController.getAfter(mouseDrag.col2-1, mouseDrag.row2+1))
            );
            // il faut qu'on ait bougé pour créer un rendez vous
            // et que la cellule de départ (ok à cause du premier if)
            // et d'arrivée soient vides
            if (mouseDrag.hasMoved()
                && Agenda.getInstance().getTableModel()
                    .getValueAt(mouseDrag.row2, mouseDrag.col2) == null)
                new RendezVousDialog(this, editorArg);
        } else if (e.getClickCount() == 2 // double clic droit pour supprimer
            && e.getButton() == MouseEvent.BUTTON3) {
            urManager.execute(new CommandRemoveRendezVous(editorArg.getRendezVous()));
            mCalendrier.repaint();
        }

        // pour éditer(ou créer) sur une cellule, il faut avoir double cliqué
        if (e.getClickCount() == 2
            && e.getButton() == MouseEvent.BUTTON1) {
            new RendezVousDialog(this, editorArg);
        }

        if (editorArg.isSubmitted()) {
            urManager.execute(new CommandAddRendezVous(editorArg.getRendezVous()));
            mCalendrier.repaint();
        }
    }
}
```

Contrôleur → Main

Enfin, la dernière classe, Main, permet de lancer l'application. Tout d'abord, elle tente de récupérer le dernier agenda ouvert. En cas d'échec, un nouvel agenda est créé.

Pour récupérer un agenda, on inspecte les arguments donnés au programme, ensuite on utilise la valeur d'une propriété permanente (stockée dans un fichier) pour retrouver le chemin du fichier. Si celui-ci est invalide, on retourne un agenda nul.

```
/**
 * Obtenir l'agenda spécifié ou récemment ouvert
 * @param args Argument contenant éventuellement l'agenda à ouvrir
 * @return L'agenda à ouvrir ou null s'il faut un nouvel agenda
 */
private static File getOrRememberAgenda(String[] args) {
    File agenda = null;
    // récupère le chemin sur la ligne de commande
    if (args.length > 0) {
        agenda = new File(args[0]);
        if (!agenda.exists()) {
            agenda = null;
            System.err.println("No such file : " + args[0]);
        }
    }
    // s'il n'y a rien dans les arguments, on lit la propriété
    // pour obtenir l'agenda récemment ouvert
    if (agenda == null) {
        String path = Main.getProperty(Main.SYSPROP_LATEST_AGENDA);
        if (!path.isEmpty()) {
            agenda = new File(path);
            if (!agenda.exists()) {
                agenda = null;
                Main.setProperty(Main.SYSPROP_LATEST_AGENDA, "");
                System.err.println("No such file : " + path);
            } else {
                Agenda.getInstance().loadFrom(agenda);
                Main.setProperty(Main.SYSPROP_LATEST_AGENDA, agenda.getAbsolutePath());
            }
        }
    }
    return agenda;
}
```

Ce qui résume le lancement à une seule instruction, rêve de tout développeur...

```
public static void main(String[] args) {
    new MainFrame(getOrRememberAgenda(args));
}
```

Mon calendrier							
Fichier Editer							
Année : 2013	Lundi 7	Mardi 8	Mercredi 9	Jeudi 10	Vendredi 11	Samedi 12	Dimanche 13
Semaine : 2	0 h						
Mois de Janvier	1 h						
	2 h						
	3 h						
	4 h						
	5 h						
	6 h						
	7 h						
	8 h	Projet	Prog. lin.	Merise	Java TP	BDD TP	BDD TP
	9 h	Projet	Prog. lin.	Merise	Java TP	BDD TP	BDD TP
	10 h	Projet	Expression	Windev	XML TP	Gestion TP	
	11 h	Projet	Expression	Windev	XML TP	Gestion TP	
	12 h						
	13 h						
	14 h	Anglais	Java	Gestion	Java DS	Prog. lin.	
	15 h	Anglais	Java	Gestion	Java DS	Prog. lin.	
	16 h	XML					
	17 h	XML					
	18 h		BDD	Merise TD			
	19 h		BDD	Merise TD			
	20 h						
	21 h						
	22 h						
	23 h						
	24 h						
	25 h						
	26 h						
	27 h						
	28 h						
	29 h						
	30 h						
	31 h						
	32 h						
	33 h						
	34 h						
	35 h						
	36 h						
	37 h						
	38 h						
	39 h						
	40 h						
	41 h						
	42 h						
	43 h						
	44 h						
	45 h						
	46 h						
	47 h						
	48 h						
	49 h						
	50 h						
	51 h						
	52 h						
	53 h						
	54 h						
	55 h						
	56 h						
	57 h						
	58 h						
	59 h						
	60 h						
	61 h						
	62 h						
	63 h						
	64 h						
	65 h						
	66 h						
	67 h						
	68 h						
	69 h						
	70 h						
	71 h						
	72 h						
	73 h						
	74 h						
	75 h						
	76 h						
	77 h						
	78 h						
	79 h						
	80 h						
	81 h						
	82 h						
	83 h						
	84 h						
	85 h						
	86 h						
	87 h						
	88 h						
	89 h						
	90 h						
	91 h						
	92 h						
	93 h						
	94 h						
	95 h						
	96 h						
	97 h						
	98 h						
	99 h						
	100 h						