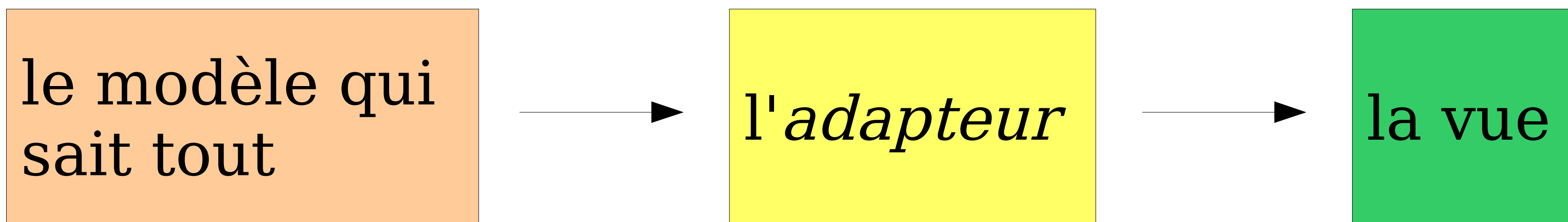
Interface Graphique en Java 1.6

Adapteurs et MVC personnalisé

Sébastien Paumier

Les adapteurs

- que faire quand on ne veut montrer qu'une partie des données d'un modèle ?
- solution: mettre un modèle intermédiaire entre le modèle qui gère toutes les données et la vue



Les adapteurs

- en pratique, l'adapteur écoute tous les changements du modèle, en tient compte, et indique ses propres changements à la vue si nécessaire
- aux yeux de la vue (qui ne connaît pas le vrai modèle), l'adapteur est le modèle
- les contrôleurs peuvent agir soit sur l'adapteur soit sur le vrai modèle

Les adapteurs

- exemple: filtrage des victimes du cours précédent en fonction de leur état
- on crée un nouveau modèle de liste qui prend en paramètre le vrai modèle ainsi qu'une valeur d'état

```
public class FilteredVictimListModel extends DefaultListModel {

    /* The real model */
    private final VictimListModel model;

    /* This adapter will only show victims that have the following state */
    private State state;

    public FilteredVictimListModel(VictimListModel model,Victim.State state) {
        if (model==null) {
            throw new NullPointerException("Invalid null VictimListModel");
        }
        this.model=model;
        this.state=state;
        register();
    }
...
}
```

Les adapteurs

- on s'enregistre sur le vrai modèle, afin d'être tenu au courant des modifications:

```
/**
 * We listen to changes that occur on the real model.
 */
private void register() {
    model.addListDataListener(new ListDataListener() {
        @Override public void contentsChanged(ListDataEvent e) {
            updateAll();
        }

        @Override public void intervalAdded(ListDataEvent e) {
            updateAll();
        }

        @Override public void intervalRemoved(ListDataEvent e) {
            updateAll();
        }
    });
}
```

Les adapteurs

- la méthode **updateAll** effectue un rafraîchissement violent de toute la liste filtrée:

```
/**
 * This is a very brute force refresh.
 */
void updateAll() {
    int oldSize=size;
    /* We update the size */
    size=-1;
    getSize();
    if (oldSize>0) {
        fireIntervalRemoved(this,0,oldSize-1);
    }
    if (size>0) {
        fireIntervalAdded(this,0,size-1);
    }
}
```

- dans une version professionnelle, il faudrait une gestion beaucoup plus fine des modifications

Les adapteurs

- on filtre avec une méthode:

```
/**
 * Returns true if the given victim is accepted by the state filter.
 */
private boolean accept(Victim v) {
    return state==null || v.getState()==state;
}
```

- et on l'utilise, notamment pour **getSize**:

```
/* We keep a cached version of the size in order to avoid
 * computations */
int size=-1;

@Override  public int getSize() {
    if (size==-1) {
        int n=model.getSize();
        size=0;
        for (int i=0;i<n;i++) {
            if (accept((Victim)model.getElementAt(i))) { size++; }
        }
    }
    return size;
}
```

Les adapteurs

- pour modifier l'état d'une victime, on doit convertir son index relatif à la liste filtrée en un index global utilisable par le vrai modèle:

```
public void upgradeVictim(int index) {
    int n=model.getSize();
    int x=-1;
    for (int i=0;i<n;i++) {
        Victim v=(Victim)model.getElementAt(i);
        if (accept(v)) {
            x++;
            if (x==index) {
                /* i is the index of the victim in the real model */
                model.upgradeVictim(i);
            }
        }
    }
}
```

Les adapteurs

- même principe pour accéder à un élément à partir de son index relatif à la liste filtrée:

```java
/**
 * We return an unmodifiable version of the requested Victim, in order
 * to be sure that the model is the only object that can modify its data.
 */
@Override
public Object getElementAt(int index) {
	int n=model.getSize();
	int x=-1;
	for (int i=0;i<n;i++) {
		Victim v=(Victim)model.getElementAt(i);
		if (accept(v)) {
			x++;
			if (x==index) return v; /* v is already a read-only Victim */
		}
	}
	return null;
}
```

Les adapteurs

- on laisse l'ajout d'un élément en fin de liste aux bons soins du vrai modèle:

```
/**
 * We delegate the insertion to the real model.
 */
@Override
public void addElement(Object o) {
    model.addElement(o);
}
```

- ce ne serait pas aussi facile si l'on insérait à un index relatif à la liste filtrée!

Les adapteurs

- enfin, quand on modifie les paramètres de l'adapteur (ici la valeur de filtre), on n'oublie pas de rafraîchir:

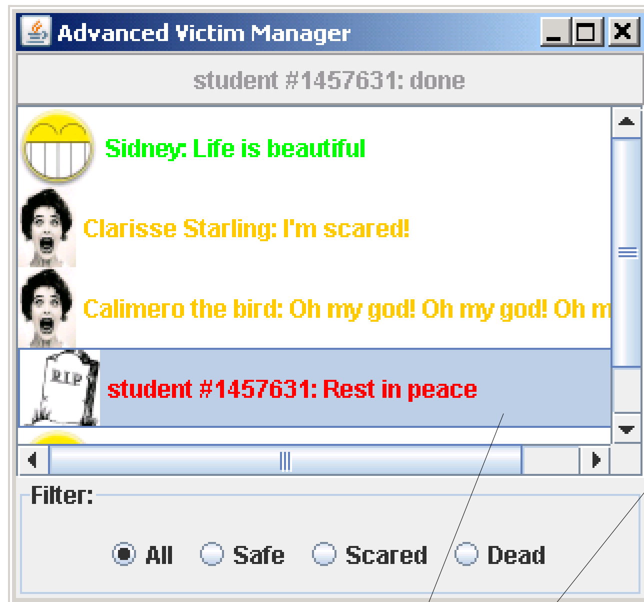
```
public void setFilter(Victim.State state) {
    this.state=state;
    updateAll();
}
```

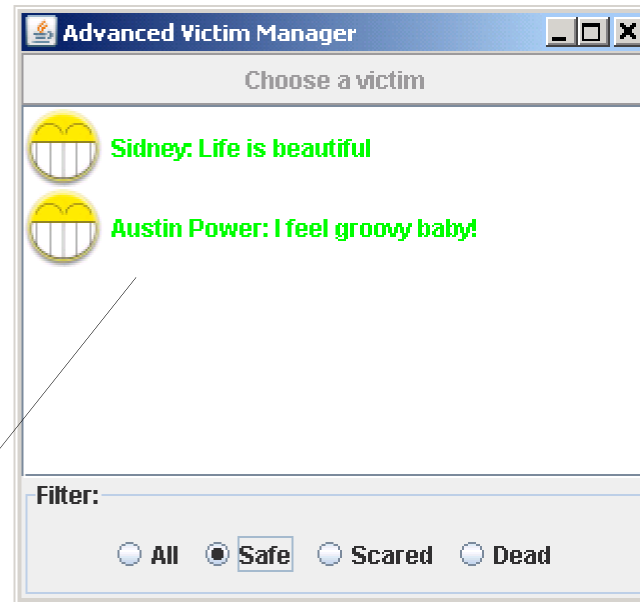
- notre adapteur est maintenant prêt, il suffit de l'utiliser:

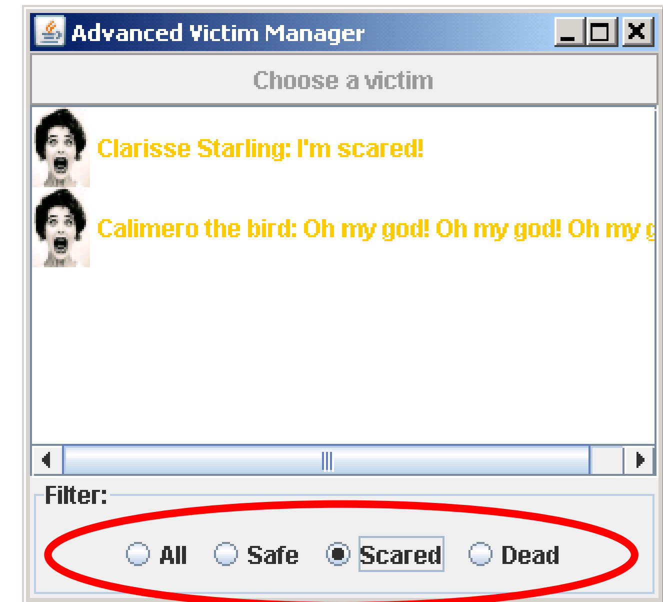
```
final VictimListModel model=new VictimListModel();
model.addElement(new Victim("Sidney"));
...
final FilteredVictimListModel adapter=new FilteredVictimListModel(model,null);
final JList list=new JList(adapter);
```

Les adapteurs

- et voilà le résultat:





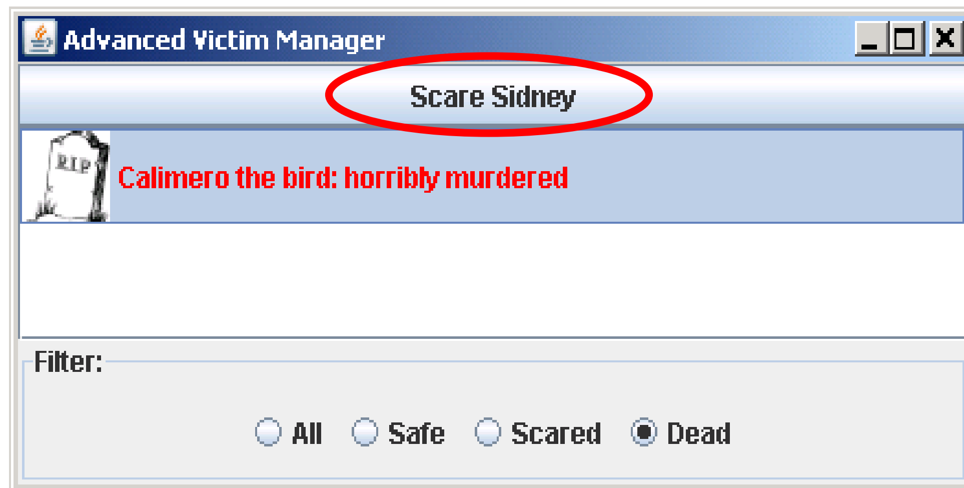


la liste se rafraîchit
comme il faut

contrôleurs agissant sur l'adapteur

Les adapteurs

- bug classique: le bouton est mal rafraîchi, car il travaille toujours sur le vrai modèle:



- facile à corriger: `model` → `adapter`

```
...
adapter.upgradeVictim(n);
...
Victim v=(Victim) adapter.getElementAt(list.getSelectedIndex());
...
```

Modification des données

- on peut également ne pas vouloir filtrer mais modifier (en apparence) les données
- exemple: le `JuryListModel`

Jury Simulator

Martin Tamarre	2.0/20
Pierre Quirroule	7.5/20
Mustapha Yo	20.0/20
Annie Toutanbloc	8.5/20
Chen de Caractaire	12.0/20

Filter: (•) Raw notes ( ) M.Z. ( ) J.C. ( ) S.P.

Jury Simulator

Martin Tamarre	6.0/20
Pierre Quirroule	11.5/20
Mustapha Yo	24.0/20
Annie Toutanbloc	12.5/20
Chen de Caractaire	16.0/20

Filter: ( ) Raw notes (•) M.Z. ( ) J.C. ( ) S.P.

Jury Simulator

Martin Tamarre	15.0/20
Pierre Quirroule	15.0/20
Mustapha Yo	15.0/20
Annie Toutanbloc	15.0/20
Chen de Caractaire	15.0/20

Filter: ( ) Raw notes ( ) M.Z. ( ) J.C. (•) S.P.

Modification des données

- cet adapteur n'agit que sur l'objet renvoyé par **getElementAt**:

```
@Override
public Object getElementAt(int index) {
    Note note=(Note) model.getElementAt(index);
    if (filter==null) return note;
    return filter.filterNote(note);
}

@Override
public int getSize() {
    return model.getSize();
}
```

- en utilisant un **ProfessorFilter**:

```
public interface ProfessorFilter {

    public Note filterNote(Note note);

}
```

Modification des données

- côté événements, l'adapteur ne fait que relayer ceux du vrai modèle:

```java
public JuryListModel(ListModel realNotes,ProfessorFilter filter) {
    this.model=realNotes;
    this.filter=filter;
    model.addListDataListener(new ListDataListener() {
        @Override public void contentsChanged(ListDataEvent e) {
            contentsChanged(e);
        }
        @Override public void intervalAdded(ListDataEvent e) {
            intervalAdded(e);
        }
        @Override public void intervalRemoved(ListDataEvent e) {
            intervalRemoved(e);
        }
    });
}

void contentsChanged(ListDataEvent e) {
    fireContentsChanged(this,e.getIndex0(),e.getIndex1());
}
void intervalAdded(ListDataEvent e) {
    fireIntervalAdded(this,e.getIndex0(),e.getIndex1());
}
void intervalRemoved(ListDataEvent e) {
    fireIntervalRemoved(this,e.getIndex0(),e.getIndex1());
}
```

`fire...` étant `protected`, on ne peut pas l'appeler depuis le `ListDataListener`

D'autres adapteurs

- autre rôle d'un adapteur: mettre des données sous une forme qui convient à un autre type de vue
- exemple: vue des victimes sous forme de table
- une **`JTable`** a besoin d'un **`TableModel`** et nous ne disposons que d'un **`ListModel`**
- solution: créer un adapteur qui soit un **`TableModel`**

Le VictimTableModel

- on étend **AbstractTableModel**, et on prend le vrai modèle en paramètre:

```
public class VictimTableModel extends AbstractTableModel {

    private final VictimListModel model;

    public VictimTableModel(VictimListModel model) {
        if (model==null) {
            throw new NullPointerException("Invalid null VictimListModel");
        }
        this.model=model;
        register();
    }
...
}
```

Le VictimTableModel

- comme tout adapteur, celui-ci écoute les changements du modèle sous-jacent:

```
private void register() {
    model.addListDataListener(new ListDataListener() {
        @Override
        public void contentsChanged(ListDataEvent e) {
            fireTableRowsUpdated(e.getIndex0(),e.getIndex1());
        }

        @Override
        public void intervalAdded(ListDataEvent e) {
            fireTableRowsInserted(e.getIndex0(),e.getIndex1());
        }

        @Override
        public void intervalRemoved(ListDataEvent e) {
            fireTableRowsDeleted(e.getIndex0(),e.getIndex1());
        }
    });
}
```

on adapte les événements à ce que la vue est capable d'entendre

Le VictimTableModel

- notre adapteur doit juste implémenter les 3 méthodes non gérées par **AbstractTableModel**:

```
@Override public int getColumnCount() {
    /* 3 columns: name, comment, state */
    return 3;
}

@Override public int getRowCount() {
    return model.getSize();
}

@Override public Object getValueAt(int rowIndex, int columnIndex) {
    Victim v=model.getElementAt(rowIndex);
    switch (columnIndex) {
    case 0: return v.getName();
    case 1: return v.getComment();
    case 2: return v.getState();
    }
    throw new AssertionError("Invalid column index: "+columnIndex);
}
```

Le VictimTableModel

- et voilà le résultat:

Ultra Advanced Victim Manager

Austin Power: done

Sidney: Oh my god! Oh my god! Oh my god!

Clarisse Starling: Everybody is cool

Calimero the bird: Life is beautiful

student #1457631: You'll never get me!

Austin Power: horribly murdered

Sidney	Oh my god! Oh my...	SCARED
Clarisse Starling	Everybody is cool	SAFE
Calimero the bird	Life is beautiful	SAFE
student #1457631	You'll never get me!	SCARED
Austin Power	horribly murdered	DEAD

Filter:

All Safe Scared Dead

cf. cours sur les tables pour la rendre plus jolie :)

Un MVC personnalisé

- on peut fabriquer entièrement un MVC
- exemple: le jeu Shangai

Les bonnes questions

- question 1: quelles sont les données ?
  - un ensemble à 3 dimensions de tuiles
- question 2: quelles sont les opérations que l'on doit pouvoir faire dessus ?
  - avoir les dimensions de l'ensemble
  - accéder à une tuile
  - savoir si une tuile est sélectionnable
  - supprimer une tuile

Les bonnes questions

- question 3: de quoi la vue doit-elle être prévenue?
  - du fait qu'une tuile est sélectionnée ou non
  - du fait qu'une tuile est supprimée

Un modèle personnalisé

- avec toutes ces informations, on peut définir notre modèle:

```
public interface ShangaiModel {

    public int getSizeX();
    public int getSizeY();
    public int getSizeZ();
    public Tile getTile(int x,int y,int z);
    public void removeTile(Tile t);
    public boolean isSelectable(Tile tile);
    public void addTileListener(TileListener tileListener);
    public void removeTileListener(TileListener tileListener);

}
```

Un modèle personnalisé

- ainsi que notre listener et notre type d'événements:

```
public interface TileListener {

    public void tileChanged(TileEvent e);

}
```

```
public class TileEvent {

private final Tile tile;
private final int type;

public final static int TILE_REMOVED=0;
public final static int TILE_STATE_CHANGED=1;

public TileEvent(Tile tile,int eventType) {
    if (tile==null) {
        throw new NullPointerException(
            "Cannot raise TileEvent for null Tile");
    }
    if (eventType!=TILE_REMOVED
        && eventType!=TILE_STATE_CHANGED) {
        throw new IllegalArgumentException(
            "Bad TileEvent type: "+eventType);
    }
this.tile=tile;
this.type=eventType;

}

public int getEventType() { return type; }

public Tile getTile() { return tile; }

}
```

Un modèle personnalisé

- cependant, **add**/**removeTileListener** et **isSelectable** seront identiques pour toutes les implémentations
- on crée donc une version abstraite du modèle qui fait tout ça:

```
public abstract class AbstractShangaiModel implements ShangaiModel {

    @Override public boolean isSelectable(Tile tile) {
        ...
    }

    private final ArrayList<TileListener> tileListeners=new ArrayList<TileListener>();

    @Override public void addTileListener(TileListener l) {
        tileListeners.add(l);
    }

    @Override public void removeTileListener(TileListener l) {
        if (firing)
            throw new IllegalStateException("Cannot remove listeners while fire... is using them");
        tileListeners.remove(l);
    }
...
```

Un modèle personnalisé

- on y met également les **fire...**:

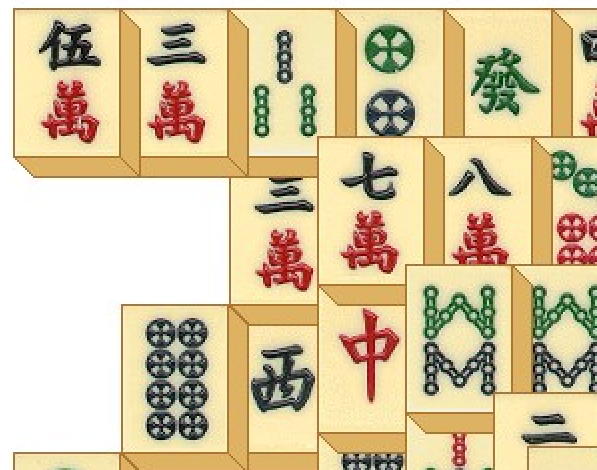
```
public abstract class AbstractShangaiModel implements ShangaiModel {

...

       protected void fireTileStateChanged(Tile t) {
        TileEvent event=new TileEvent(t, TileEvent.TILE_STATE_CHANGED);
        try {
            firing = true;
            for (TileListener l : tileListeners) {
                l.tileChanged(event);
            }
        } finally {
            /*
             * Very important to avoid being blocked because of an exception
             * raised in a listener implementation
             */
            firing = false;
        }
    }

    protected void fireTileRemoved(Tile t) {
        ...
    }

}
```

Un modèle personnalisé

- il ne nous reste plus qu'à implémenter **`ShangaiModelImpl`** qui va:
  - implémenter les méthodes du modèle qui ne sont pas gérées par **`AbstractShangaiModel`**
  - s'initialiser en disposant les tuiles comme il faut (en tortue)
  - dire s'il reste des coups jouables
  - pouvoir mélanger les tuiles restantes en cas de blocage
  - etc

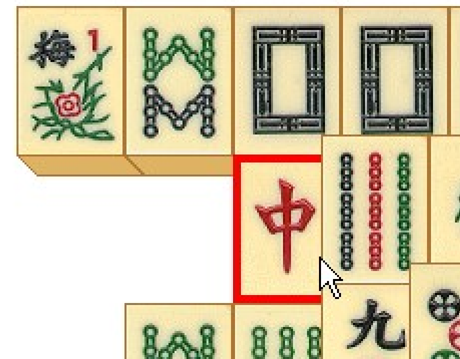
Une vue personnalisée

- comment afficher le jeu ?
  - avec un `JLayeredPane` contenant des couches de tuiles
- comment dessiner une couche ?
  - en dessinant les tuiles dans le bon ordre pour éviter ça:



Une vue personnalisée

- quand rafraîchir la vue ?
  - chaque couche doit se rafraîchir quand une de ses tuiles est modifiée
- quand on sélectionne une tuile, on va juste modifier son aspect supérieur, donc pas besoin de tout redessiner



Une vue personnalisée

- pour chaque couche, on va garder une image qui la représente
- quand une tuile est (dé)sélectionnée, on aura juste la zone de la tuile à repeindre
- on reconstruira tout seulement en cas de suppression de tuile

Une vue personnalisée

- voici donc le constructeur d'une couche:

```
public TilePane(ShangaiModel model, final int layer) {
    this.model = model;
    this.layer = layer;
    setOpaque(false);
    bufferedImage = new BufferedImage(ShangaiPane.GAME_PANE_WIDTH,
            ShangaiPane.GAME_PANE_HEIGHT, BufferedImage.TYPE_INT_ARGB);
    paintTilePane();
    model.addTileListener(new TileListener() {
        @Override
        public void tileChanged(TileEvent e) {
            Tile t=e.getTile();
            if (t.getZ() == layer) {
                if (e.getEventType()==TileEvent.TILE_STATE_CHANGED) {
                    paintPartiallyTilePane(e.getTile());
                } else {
                    /* We repaint all the pane if a tile was removed */
                    paintTilePane();
                }
            }
        }
    });
}
```

cette vue partielle s'enregistre sur le modèle pour se tenir à jour

Une vue personnalisée

- quant à la vue globale, son seul travail consiste à attraper les clics et à jouer le rôle de contrôleur quand une tuile a été cliquée:

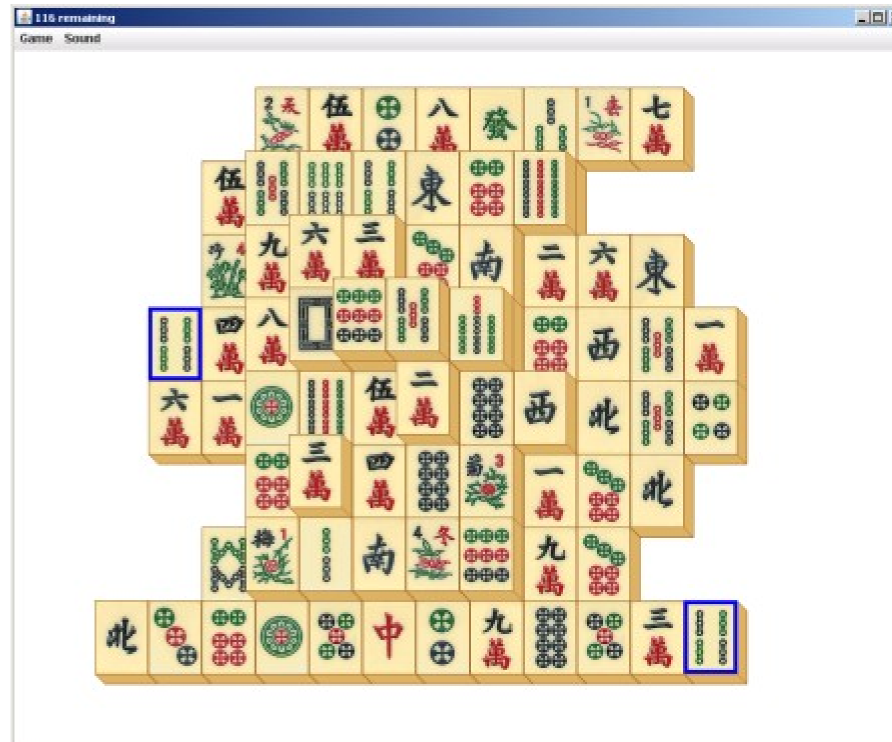
```
public ShangaiPane(final ShangaiModelImpl model) {
    this.model = model;
    setPreferredSize(new Dimension(GAME_PANE_WIDTH,GAME_PANE_HEIGHT));
    JPanel background = new JPanel();
    background.setBackground(Color.WHITE);
    background.setOpaque(true);
    background.setBounds(0, 0, GAME_PANE_WIDTH, GAME_PANE_HEIGHT);
    add(background, new Integer(0));
    for (int i = 0; i < model.getSizeZ(); i++) {
        TilePane p = new TilePane(model, i);
        p.setBounds(0, 0, GAME_PANE_WIDTH, GAME_PANE_HEIGHT);
        add(p,Integer.valueOf(1+i));
    }
    addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent e) {
            model.setClickedTile(tileClicked(e.getX(), e.getY()));
        }
    });
}
```

Une vue personnalisée

- c'est à l'implémentation du modèle qu'il revient de savoir quoi faire quand on a cliqué sur une tuile:
  - la tuile cliquée est-elle sélectionnable ?
    - si non, erreur
    - si oui, y avait-il déjà une tuile sélectionnée ?
      - si non, sélectionner la tuile courante
      - si oui, les 2 tuiles sont-elles compatibles ?
        - si non, erreur
        - si oui, on les enlève
          - la partie est-elle finie ?
          - la partie est-elle bloquée ?

Un MVC personnalisé

- on a donc tout ce qu'il faut pour écrire l'application complète:



- enjoy !