

# **Programmation orientée objet**

Introduction aux interfaces  
graphiques

# Introduction : Programmation Séquentielle vs. Événementielle

---

- **Programmation séquentielle**

La séquence des opérations effectuées est prédéterminée. Le programme dicte le moment où l'utilisateur doit entrer de l'information, et le moyen par lequel il peut le faire.

- **Programmation événementielle**

La séquence des opérations effectuées est variable. Le programme réagit aux « actions » de l'utilisateur dont la séquence est à la fois inconnue et imprévisible.

# Programmation Séquentielle

---

- En programmation séquentielle, le programme interroge directement l'utilisateur. S'il désire utiliser le programme, l'utilisateur doit se laisser « contrôler ».
- Exemple :

```
int val1, val2;  
cout << "Entrer un premier nombre :" << endl;  
cin >> val1;  
cout << "Entrer un deuxieme nombre :" << endl;  
cin >> val2;  
cout << "La moyenne est de " << (val1 + val2)/2.0;
```

# Programmation Séquentielle

## (suite)

---

- En programmation séquentielle, on ne traite qu'**une seule source d'entrées** à la fois (par exemple le clavier).
- Or, avec le développement d'interfaces graphiques complexes, les sources d'entrées se sont multipliées (clavier, souris, stylet, manette, écran tactile, caméras, etc.)
- La programmation séquentielle n'est pas appropriée à l'utilisation d'interfaces complexes puisqu'elle devrait connaître d'avance le moment et la provenance des entrées.

# Événement (définition)

---

- En français :

Un **événement** est un fait qui survient à un moment donné. Il se caractérise par une transition, voire une rupture, dans le cours des choses, et par son caractère relativement soudain ou fugace, même s'il peut avoir des répercussions par la suite. Au sens général, il signifie tout ce qui arrive et possède un caractère peu commun, voire exceptionnel.

- En informatique :

Un événement est un message indiquant que **quelque chose s'est produit**. Celui-ci peut, ou non, engendrer une série d'opérations visant à réagir au changement d'état.

# Programmation événementielle

---

- La programmation événementielle consiste à écrire un programme qui répond (ou réagit) aux différents événements à mesure que ceux-ci surviennent dans le système.
- Un événement peut être par exemple :
  - Un clic de souris
  - Une touche (ou une combinaison de touches) enfoncée sur le clavier
  - La réception d'un paquet réseau
  - L'arrivée à zéro d'un compteur à rebours (*timer*).

# Séquentielle vs. Événementielle par analogie simpliste

## PROGRAMMATION SEQUENTIELLE

On va prendre un verre tout de suite?

Nope.

On va prendre un verre tout de suite?

Nope.

On va prendre un verre tout de suite?

Nope.

On va prendre un verre tout de suite?

Nope.



# Séquentielle vs. Événementielle par analogie simpliste

## PROGRAMMATION SEQUENTIELLE

On va prendre un verre tout de suite?

Nope.

On va prendre un verre tout de suite?

Nope.

On va prendre un verre tout de suite?

Nope.

On va prendre un verre tout de suite?

Nope.



2015

## PROGRAMMATION ÉVÉNEMENTIELLE

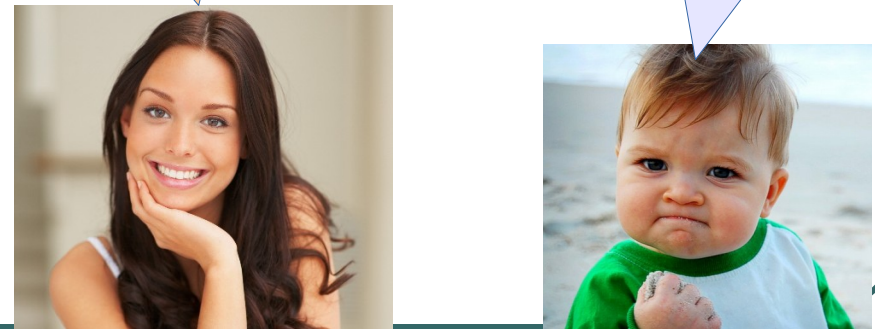
Dis-moi quand tu veux aller prendre un verre!

Ok.



Je suis prête.

Ok !



Raphaël Beamonte

8

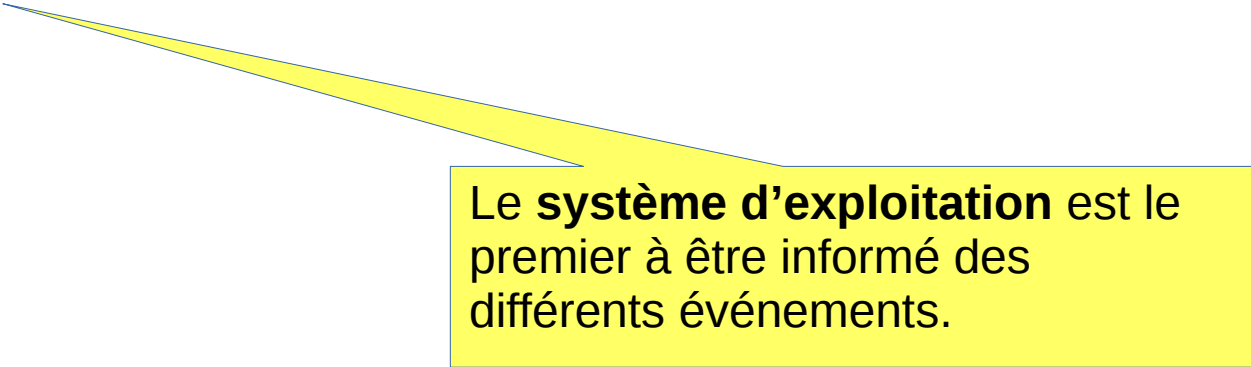


# Gestion des événements

---

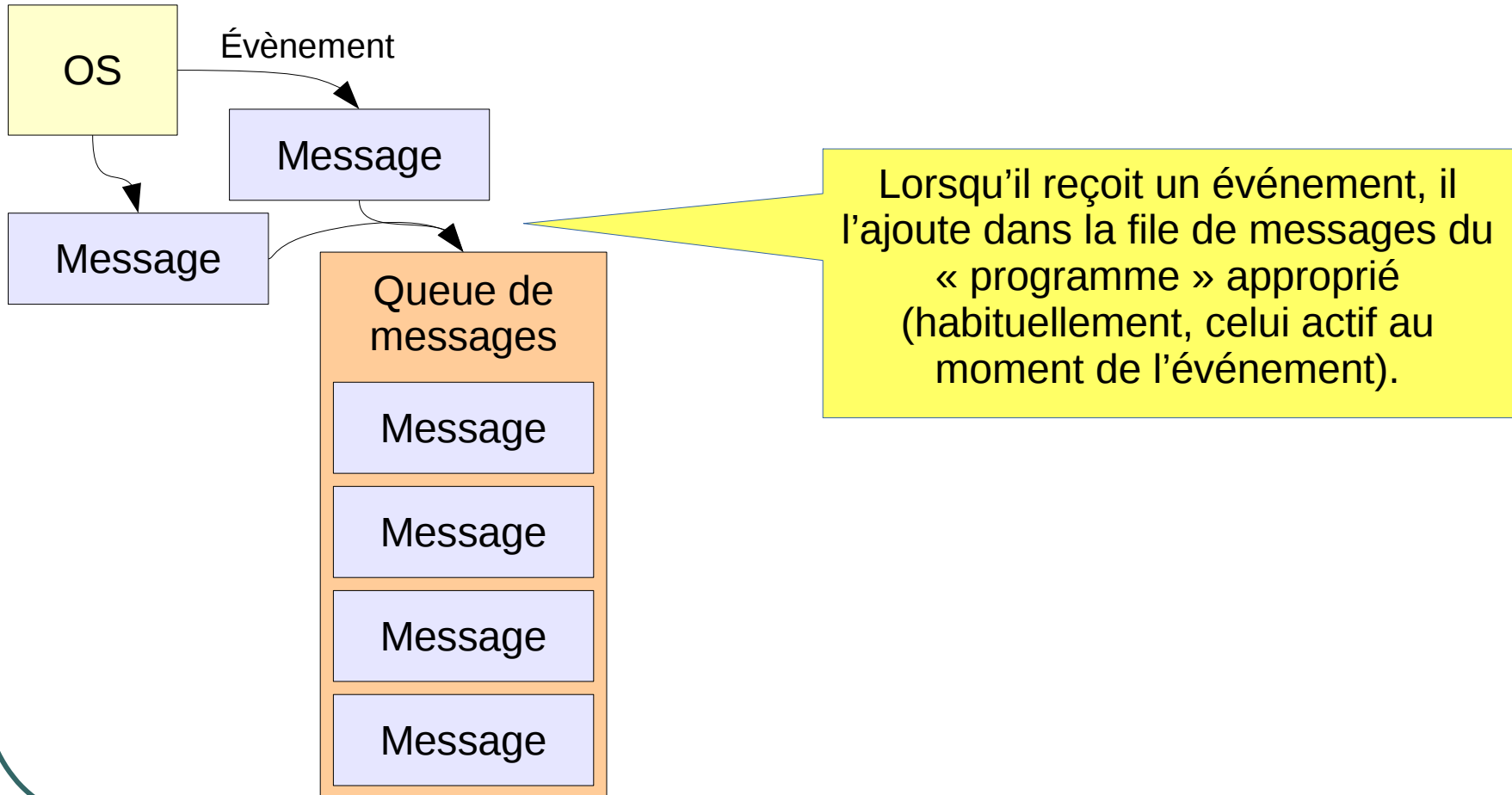
OS

Évènement

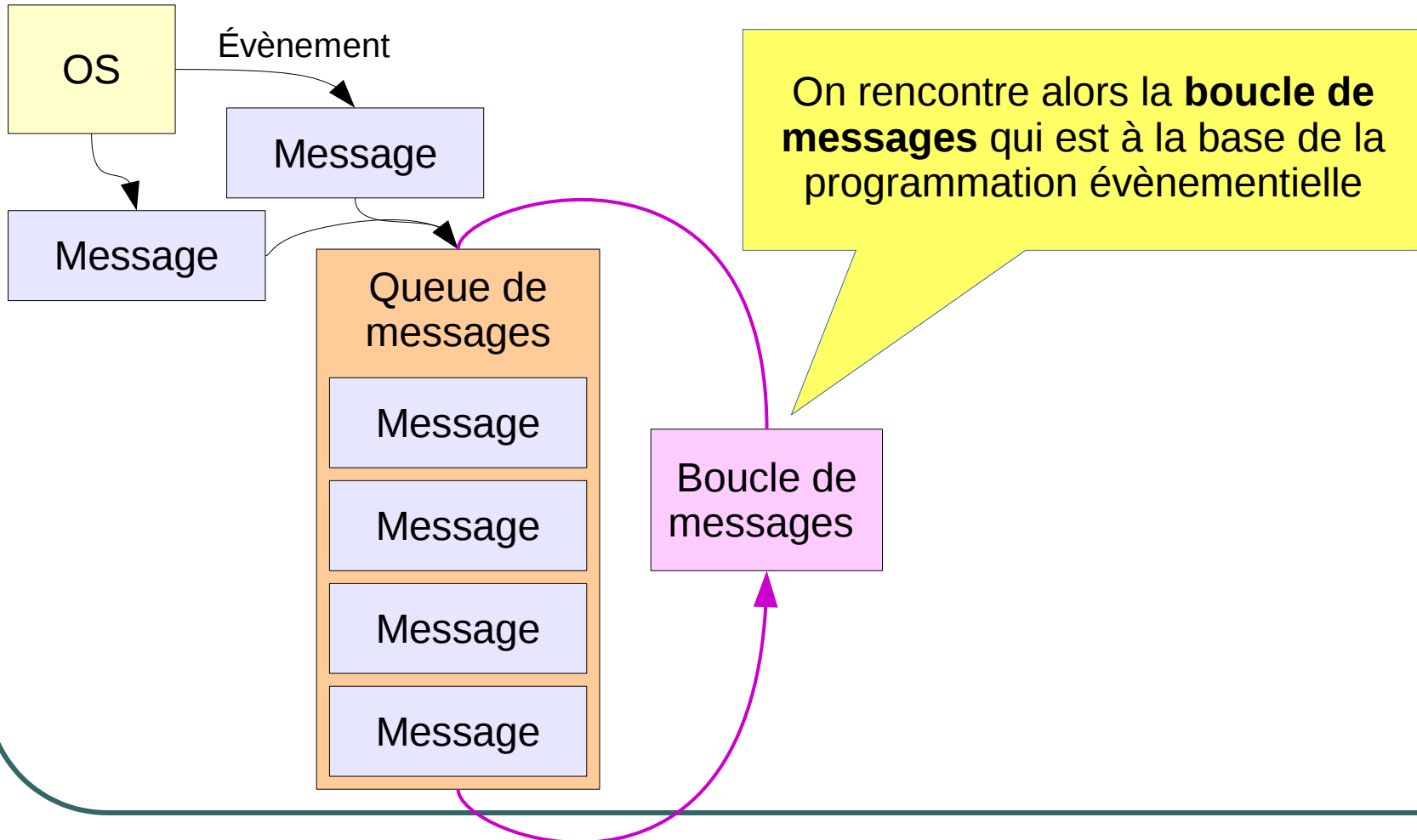


Le **système d'exploitation** est le premier à être informé des différents événements.

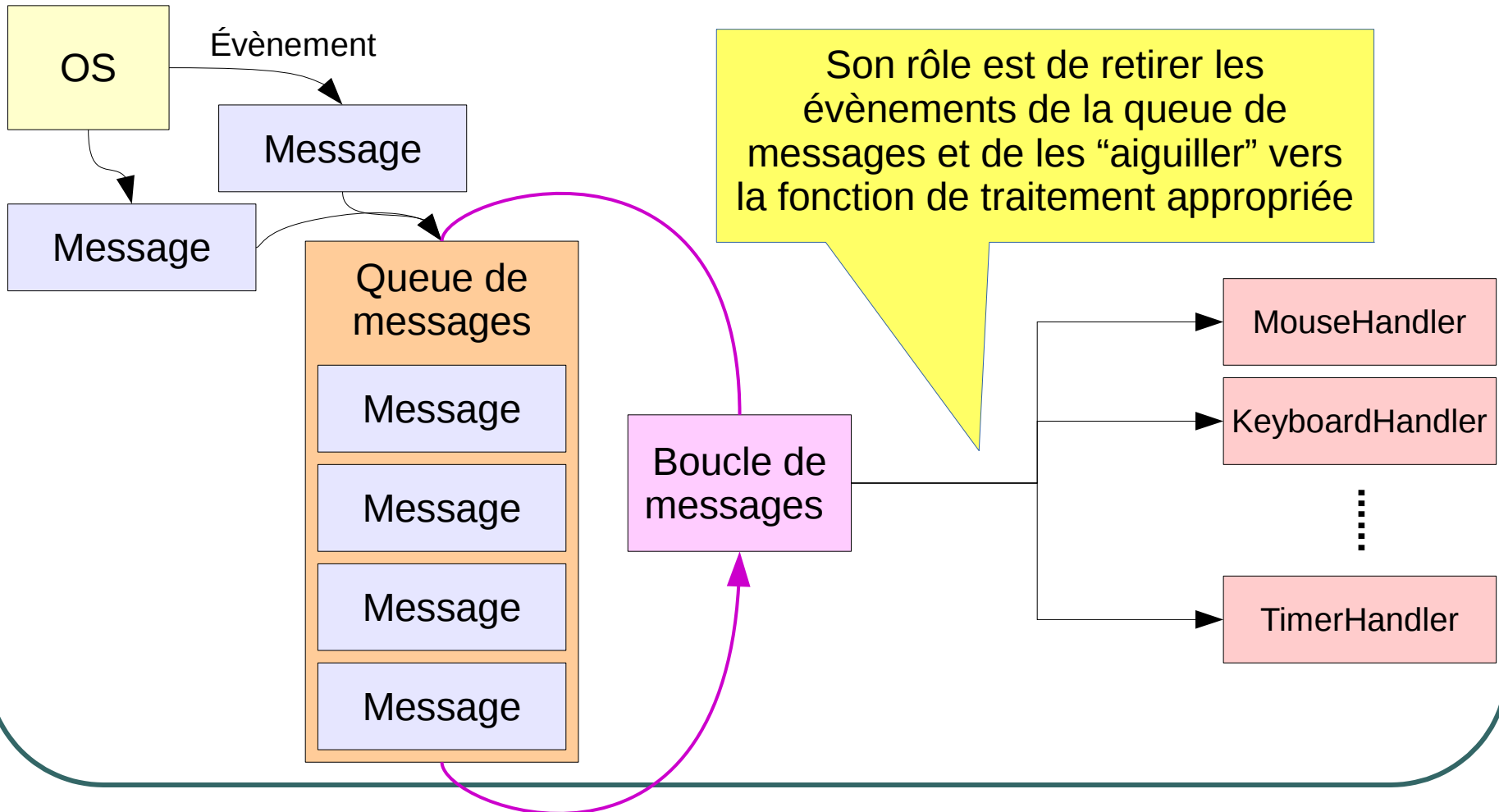
# Gestion des événements



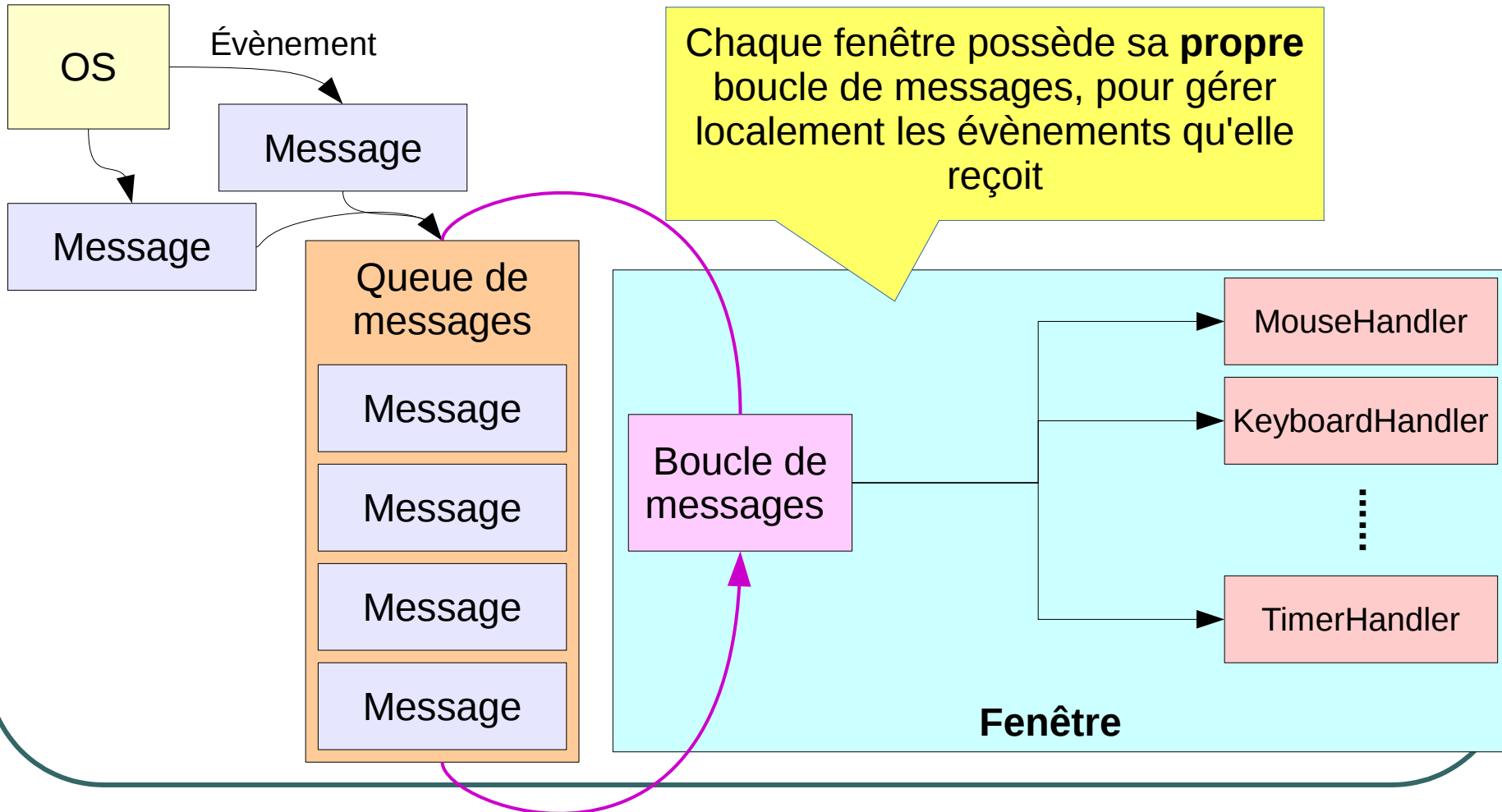
# Gestion des événements



# Gestion des événements

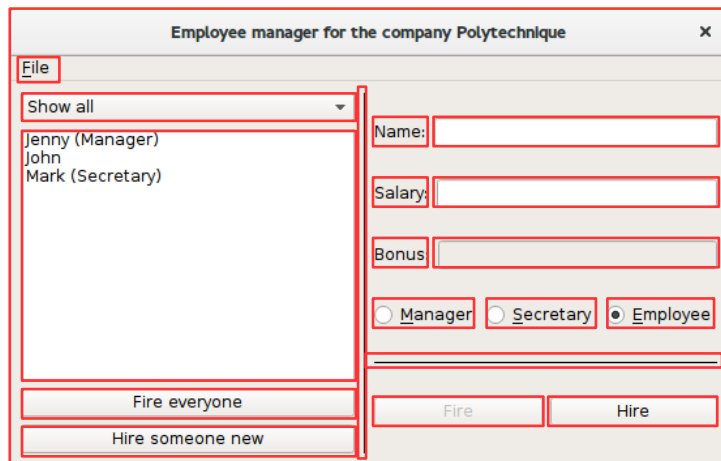


# Gestion des événements



# Éléments d'interface graphique (widgets)

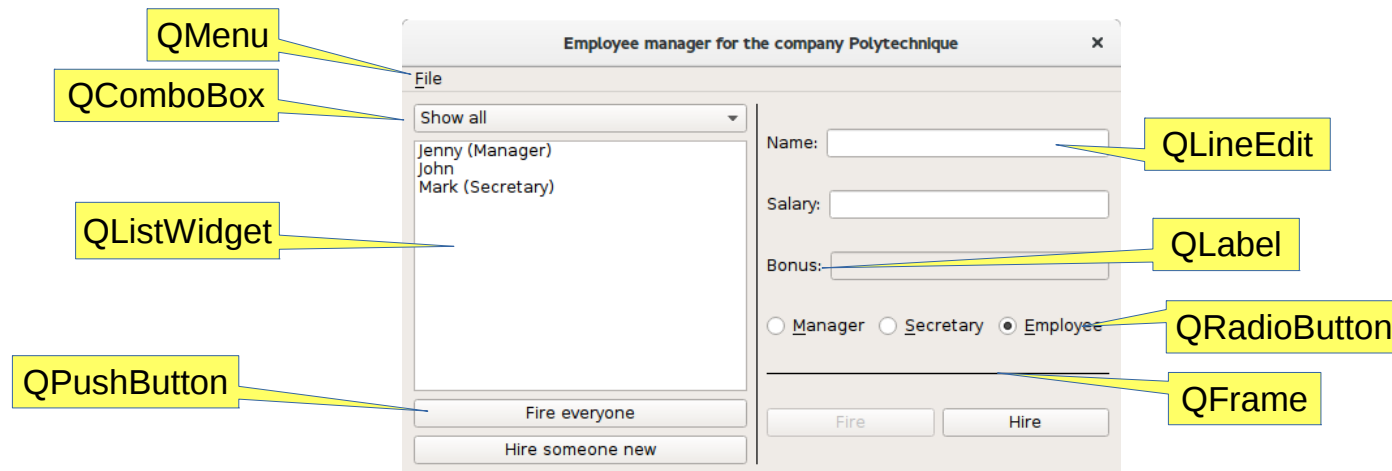
- **Tous** les éléments d'une interface graphique sont des fenêtres (ou sous-fenêtres). La façon dont ceux-ci réagissent aux différents événements (Boucle de messages) dictera leur utilité.



Tous les éléments encadrés en rouge, entre autres, sont conceptuellement des fenêtres d'interface graphique

# Éléments d'interface graphique (widgets)

- Certains éléments (*widgets*) sont tellement fréquemment utilisés (boutons, champs texte, etc.) qu'ils sont intégrés à certaines bibliothèques de développement (*Qt*, *WPF*, *Win32 API*, *wxWidgets*, *Java Swing*, etc.).



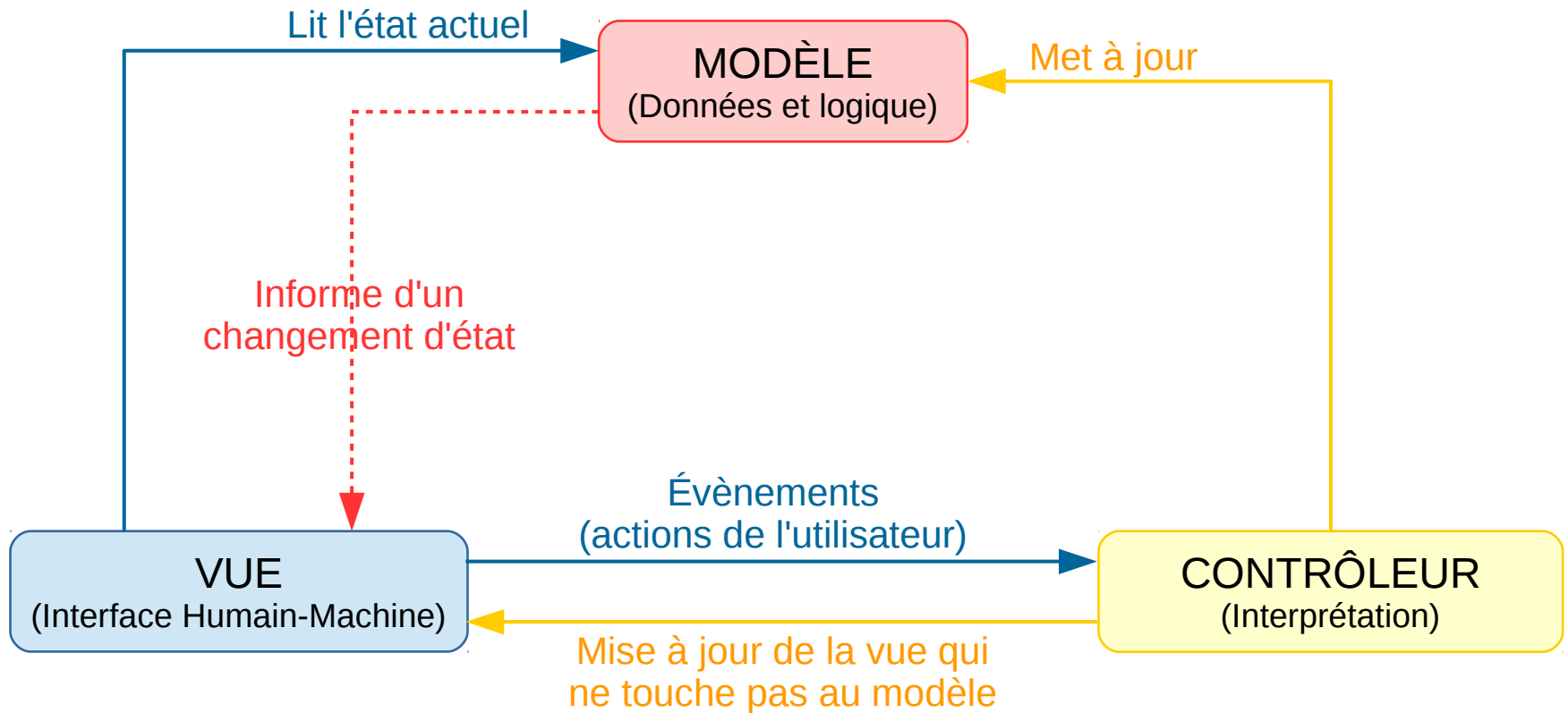
# Modèle – Vue – Contrôleur (MVC)

---

- C'est un **patron de conception** très utile en programmation événementielle et surtout dans la conception d'interfaces graphiques (GUI).
- Cela permet de **séparer les tâches en plusieurs classes pour diminuer la complexité** du programme ou du composant, et d'offrir une **grande flexibilité et réutilisabilité** du code.
- Séparation des données (modèle), de la présentation (vue), et des traitements (contrôleur).



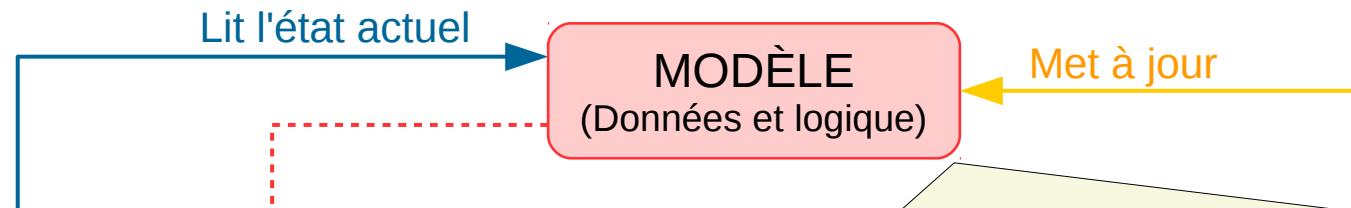
# Modèle – Vue – Contrôleur (Relations)



# Modèle – Vue – Contrôleur

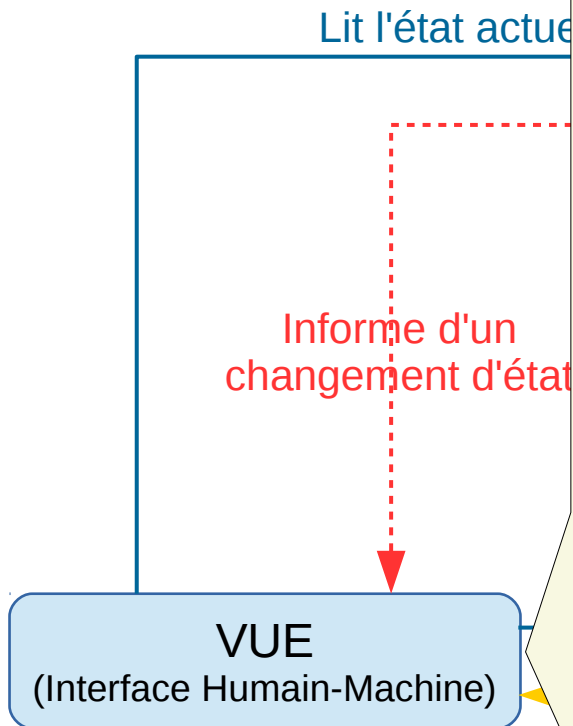
## (Relations)

---



- Le Modèle sert à:
  - Maintenir l'état des données et fournir des méthodes d'accès et de modification pour celles-ci
  - Signaler aux vues que l'état de ses attributs a été modifié
  - Gérer les événements qui ne proviennent pas de l'interface graphique (i.e. Timer)
- Le Modèle ne formate pas les données pour la sortie: c'est le rôle de la Vue.
- Il doit fournir un ensemble de méthodes assez larges pour que ses données soient modifiées puisqu'il ne connaît pas le besoin des Contrôleurs.

# Modèle – Vue – Contrôleur (Relations)



- La **Vue** est l'interface avec laquelle l'utilisateur interagit. C'est généralement le point d'entrée des événements.
- Son rôle est de:
  - Signaler au **Modèle** qu'elle veut être avertie lors de modifications des données
  - Formater les données pour l'affichage
  - Représenter les données graphiquement
  - Connaître les actions de l'utilisateur, et en rediriger les événements à son **Contrôleur** pour interprétation
- Plusieurs **Vues** peuvent afficher les données d'un même **Modèle** (i.e. un point dans un intervalle, ou la valeur correspondant)

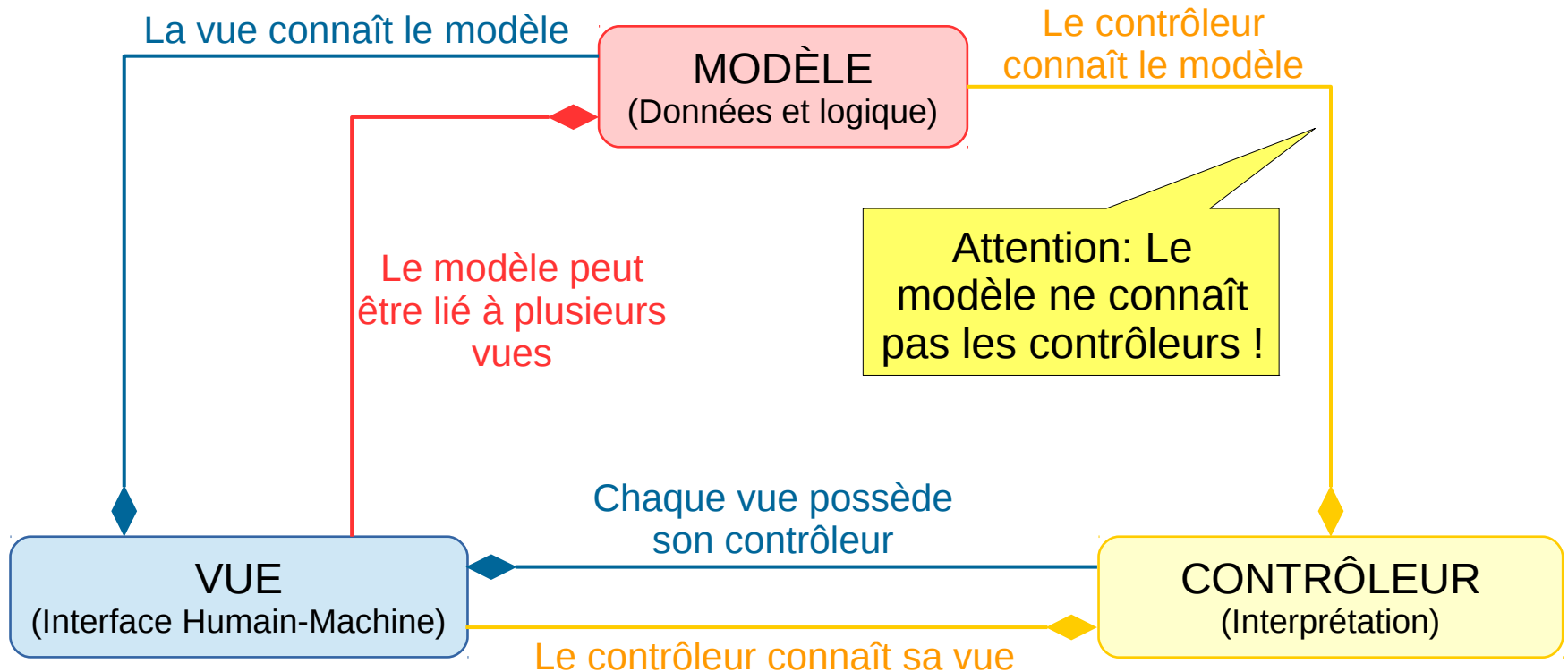
# Modèle – Vue – Contrôleur (Relations)

- Le **Contrôleur** a pour rôle de:
  - Gérer les évènements
  - Interpréter les actions de l'utilisateur en un changement sur le **Modèle** ou sur sa **Vue**:
    - Si une donnée doit être modifiée, modifier le **Modèle** (qui le signalera aux **Vues**)
    - Si le changement est purement esthétique, l'appliquer à la **Vue** (i.e. couleur d'arrière plan, habillage d'un élément, etc.)
  - Vérifier la validité des actions de l'utilisateur: il ne doit autoriser un changement que s'il est permis à l'utilisateur.
- Le **Contrôleur** ne fait aucun traitement. Il ne modifie pas / n'adapte pas les données.

Met à jour

**CONTRÔLEUR**  
(Interprétation)

# Modèle – Vue – Contrôleur (Liens)



# La bibliothèque graphique Qt : quel est son intérêt ?

---

- C'est une bibliothèque multiplateformes (GNU/Linux, Windows, Mac OS X, Android, iOS, WinRT, etc.) largement utilisé pour la conception d'applications graphiques.
- **Qt** est notamment connu pour être la bibliothèque sur laquelle repose KDE, l'un des environnements de bureau les plus utilisés dans le monde GNU/Linux.
- Il offre environ 500 classes d'éléments graphiques, conteneurs génériques, outils de modification de texte, dessins 2D, etc.
- Il simule l'aspect et la convivialité des applications natives des systèmes d'exploitation (*look and feel*).



# La bibliothèque graphique Qt

---

- Bon, par contre... **Qt** n'utilise pas vraiment MVC, mais une version simplifiée : l'architecture **Modèle-Vue**.



- En fait, dans ce modèle, le **Contrôleur** est intégré à la **Vue**. On diminue ainsi un peu la complexité de MVC, tout en gardant les données séparées de l'affichage.

# La bibliothèque graphique Qt

## Signals & slots

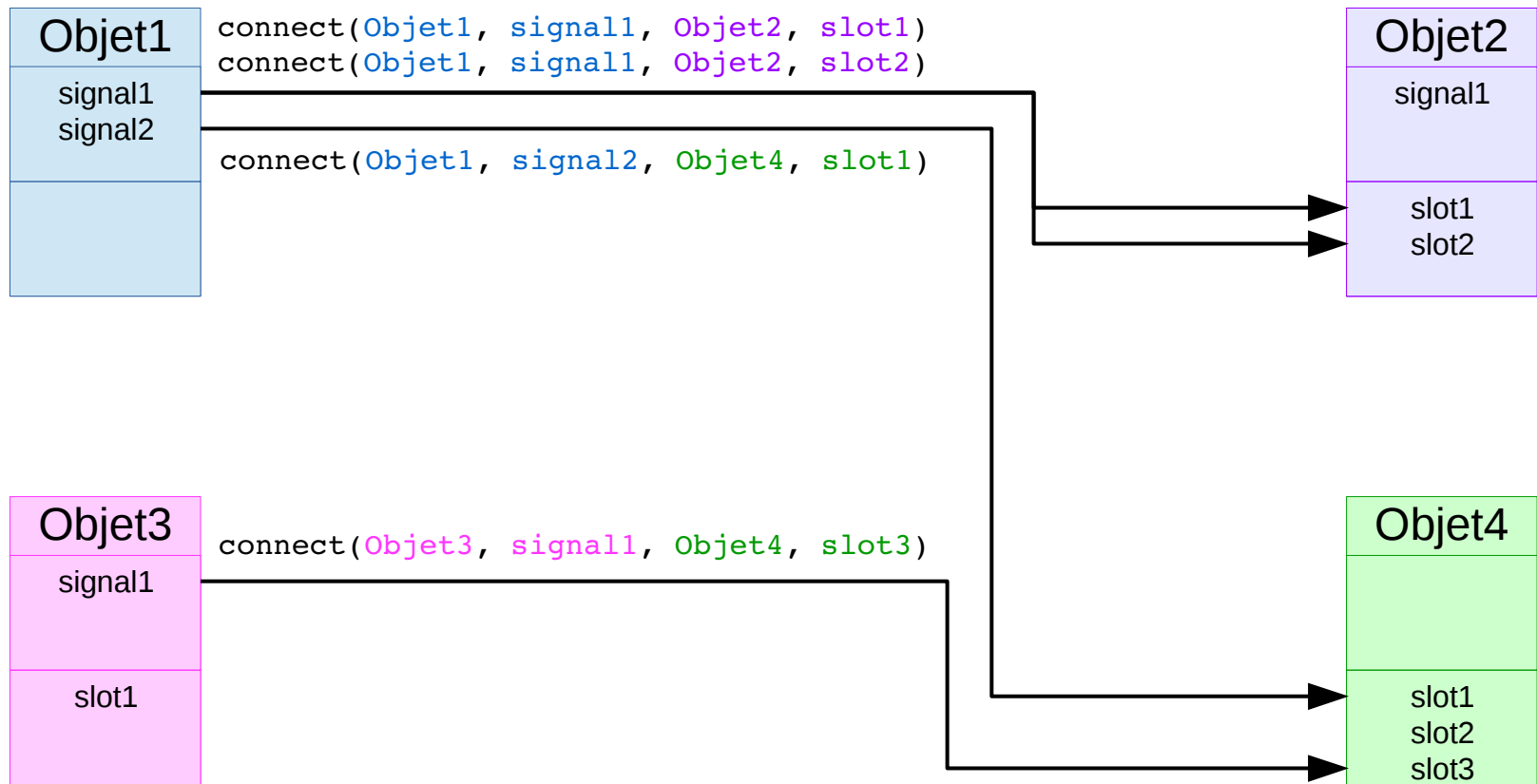
---

- Parmi les points forts de **Qt**, on trouve incontestablement le mécanisme de **signals** et **slots**.
- Ce mécanisme permet de gérer les événements au sein d'une fenêtre en liant une action (i.e. clic sur un bouton) à une réaction (i.e. fermeture de la fenêtre).
- Concrètement :
  - Un **signal** est un message envoyé lorsqu'un événement se produit.
  - Un **slot** est la fonction, généralement une méthode d'une classe, qui est appelée lorsqu'un événement s'est produit.
  - Si on relie un **signal** à un ou plusieurs **slots**, lorsque ce **signal** est émis, le ou les **slots** liés sont exécutés.



# La bibliothèque graphique Qt

## Signals & slots : exemple



# La bibliothèque graphique Qt

## Signals & slots : exemple

---

- Si on considère la classe minimale suivante :

```
class Counter
{
public:
    Counter() { val_ = 0; }
    int getValue() const { return val_; }
    void setValue(int value);

private:
    int val_;
};
```

# La bibliothèque graphique Qt

## Signals & slots : exemple

- Une version **Qt** ressemblerait à ça :

```
#include <QObject>

class Counter : public QObject
{
    Q_OBJECT
public:
    Counter() { val_ = 0; }
    int getValue() const { return val_; }

public slots:
    void setValue(int value);

signals:
    void valueChanged(int newValue);

private:
    int val_;
};
```

On inclut la classe dont on va dériver notre compteur

Notre compteur devient un objet dérivé d'un objet Qt

Cette macro permet de rendre disponibles les signaux et slots

**setValue** est devenu un **slot** : on pourra le lier à un **signal** pour que la valeur change lors de certains évènements

On ajoute un **signal** qui sera émis lorsque la valeur changera

# La bibliothèque graphique Qt

## Signals & slots : exemple

---

- Les **signals** n'ont pas d'implémentation, puisque lors de leur émission ce sont des **slots** qui seront exécutés.
- Les **slots**, par contre, doivent donc être implémentés. Pour notre **slot setValue**, on aura donc :

```
void Counter::setValue(int value)
{
    if (value != val_) {
        val_ = value;
        emit valueChanged(value);
    }
}
```

Le mot clé **emit** sert à émettre un **signal**. Ici, on émet le signal **valueChanged** lorsque la valeur est changée, avec pour paramètre la nouvelle valeur.

# La bibliothèque graphique Qt

## Signals & slots : exemple

- On peut alors connecter nos **slots** et **signals** ensemble.
- Par exemple, créons deux instances de **Counter**, et lions-les pour que lorsque l'on modifie le compteur **c1**, le compteur **c2** soit modifié, mais pas l'inverse :

```
int main() {  
    Counter c1, c2;  
    QObject::connect(&c1, SIGNAL(valueChanged(int)),  
                    &c2, SLOT(setValue(int)));  
  
    c1.setValue(42);  
    cout << "C1: " << c1.getValue()  
        << "; C2: " << c2.getValue()  
        << endl; // C1: 42; C2: 42  
  
    c2.setValue(1337);  
    cout << "C1: " << c1.getValue()  
        << "; C2: " << c2.getValue()  
        << endl; // C1: 42; C2: 1337  
}
```

On connecte le **signal** **valueChanged** de l'objet **c1** au **slot setValue** de l'objet **c2**

Lorsqu'on modifie **c1**, le signal est émis, le slot exécuté, est **c2** est aussi modifié

On n'a connecté le signal que dans un sens : rien ne se passe pour **c1** lorsqu'on modifie **c2**

# La bibliothèque graphique Qt

## Signals & slots : exemple

- On peut alors connecter nos **slots** et **signals** ensemble.
- Par exemple, créons deux instances de **Counter**, et lions-les pour que lorsque l'on modifie le compteur **c1**, le compteur **c2** soit modifié, mais pas l'inverse :

```
int main() {  
    Counter c1, c2;  
    QObject::connect(&c1, &Counter::valueChanged,  
                    &c2, &Counter::setValue);  
  
    c1.setValue(42);  
    cout << "C1: " << c1.getValue()  
        << "; C2: " << c2.getValue()  
        << endl; // C1: 42; C2: 42  
  
    c2.setValue(1337);  
    cout << "C1: " << c1.getValue()  
        << "; C2: " << c2.getValue()  
        << endl; // C1: 42; C2: 1337  
}
```

Autre écriture valide qui utilise l'adresse de la méthode au lieu des macros **SIGNAL** et **SLOT** fournies par Qt

# La bibliothèque graphique Qt

## Hello World, you're so cute.

- Un programme fenêtré peut être écrit en très peu de lignes avec **Qt**, comme le montre cet exemple basique :

```
#include <QApplication>
#include <QLabel>
```

Chaque classe possède un fichier d'entête du même nom

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
```

Tout élément de **Qt** qui ne possède pas de parent devient lui-même une fenêtre complète

```
    QLabel* hello = new QLabel("Hello World, "
                               "you're so cute.");
```

```
    hello->show();
```

Pour afficher l'élément à l'écran

```
    return app.exec();
```

```
}
```

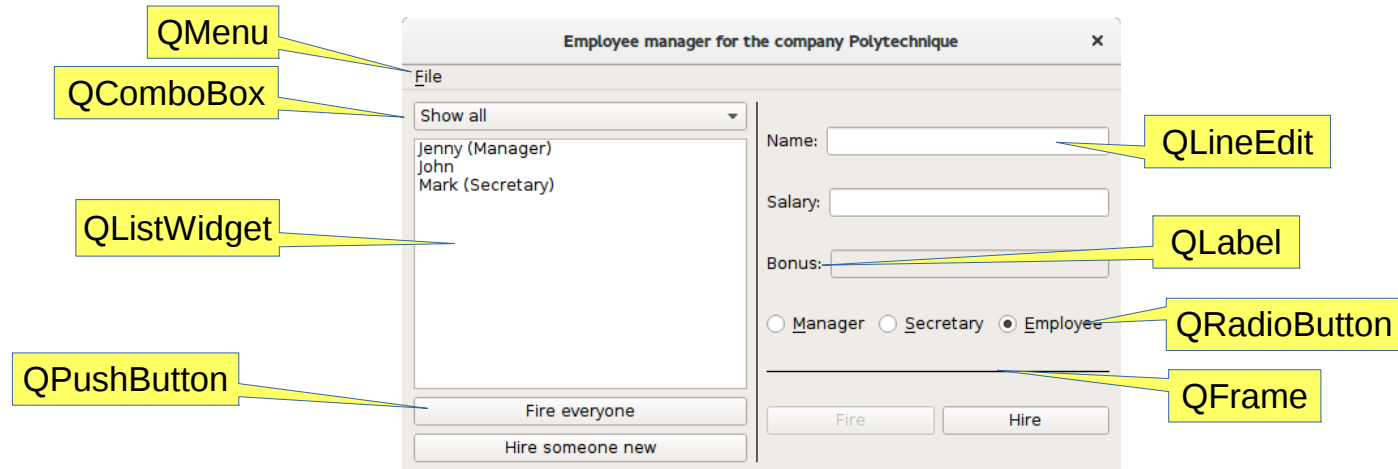
On démarre la boucle d'évènements: le programme ne se terminera que lorsque l'utilisateur aura demandé à quitter



# La bibliothèque graphique Qt

## Company

- Appliquons l'utilisation de Qt à notre programme de gestion des employés d'une entreprise. On l'a vu, on souhaite obtenir le programme suivant :



- Dans notre cas, le modèle sera la classe **Company**, et la vue sera une nouvelle classe **MainGui**



# La bibliothèque graphique Qt

## Company : la classe Company

---

- On va tout d'abord ajouter des signaux à notre modèle :

```
class Company : public QObject
{
    Q_OBJECT

    /* ... */

signals:
    /**
     * @brief employeeAdded Signal sent when an employee is added
     * @param employee The employee that has been added
     */
    void employeeAdded(Employee* employee);
    /**
     * @brief employeeDeleted Signal sent when an employee is deleted
     * @param employee The employee that has been deleted
     */
    void employeeDeleted(Employee* employee);

    /* ... */
};
```

# La bibliothèque graphique Qt

## Company : la classe Company

---

- Que l'on va émettre lorsque les situations adéquates se présentent :

```
void Company::addEmployee(Employee* employee)
{
    employees_.push_back(employee);
    emit employeeAdded(employee);
}

void Company::delEmployee(Employee* employee)
{
    auto it = find(employees_.begin(), employees_.end(), employee);

    if (it != employees_.end()) {
        Employee* e = *it;
        employees_.erase(it);
        emit employeeDeleted(e);
    }
}
```

# La bibliothèque graphique Qt

## Company : la classe MainGui

- Puis, créer notre nouvelle classe :

```
class MainGui : public QMainWindow
{
    Q_OBJECT
```

On fait dériver notre classe de **QMainWindow**, qui dérive de **QWidget**, qui elle-même dérive de **QObject**

```
public:
```

Constructeur par défaut qui créera un nouvel objet **Company**

```
    MainGui(QWidget *parent = 0);
    MainGui(Company* company, QWidget *parent = 0);
    ~MainGui();
```

Constructeur par paramètre pour préciser sur quel objet **Company** on veut travailler

```
private:
```

```
    /** @brief setup To setup the GUI */
    void setup();
    /** @brief setMenu To prepare the top menu of the GUI */
    void setMenu();
    /** @brief setUI To prepare the content of the GUI */
    void setUI();
    /** @brief loadEmployees To load the employees of the company into the QListWidget */
    void loadEmployees();
    /** @brief filterHide Function to return whether the given employee has to be hidden */
    void filterHide(Employee*);
```

```
/* ... */
```

# La bibliothèque graphique Qt

## Company : la classe MainGui

- On aura besoin de certains attributs :

Certains attributs servent à stocker des informations pour y accéder plus tard

```
/* ... */
/** @brief company_ To store the company on which the GUI works */
Company* company_;
/** @brief added_ To store the list of locally created employees */
vector<Employee*> added_;
/** @brief companyIsLocal_ To know if we created a new company locally */
bool companyIsLocal_;
/** @brief currentFilterIndex_ To store the current filter for the employees list */
int currentFilterIndex_;
/** @brief employeesList Graphical list of the employees */
QListWidget* employeesList;
/** @brief .. Graphical edit fields for the employee name/salary/bonus */
QlineEdit *nameEditor, *salaryEditor, *bonusEditor;
/** @brief employeeTypeRadioButtons List of radio buttons for the employee type */
list<QRadioButton*> employeeTypeRadioButtons;
/** @brief fireButton Graphical button to fire an employee */
QPushButton* fireButton;
/** @brief hireButton Graphical button to hire a new employee */
QPushButton* hireButton;
/* ... */
```

D'autres vont servir à stocker des éléments graphiques pour pouvoir les utiliser dans différentes méthodes de la classe

# La bibliothèque graphique Qt

## Company : la classe MainGui

---

- Finalement, on aura des **slots** :

```
/* ... */
```

```
public slots:
    /** @brief filterList Slot to filter the list according to the received parameter */
    void filterList(int);
    /** @brief selectEmployee Slot to select an employee given a QListWidgetItem */
    void selectEmployee(QListWidgetItem*);
    /** @brief cleanDisplay To clean the editor on the right of the GUI */
    void cleanDisplay();
    /** @brief changedType To update the editor when we select another type of employee */
    void changedType(int);
    /** @brief fireEveryone To fire all the employees */
    void fireEveryone();
    /** @brief fireSelected To fire only the selected employees */
    void fireSelected();
    /** @brief createEmployee To create a new employee locally */
    void createEmployee();
    /** @brief employeeHasBeenAdded To run when an employee has been added */
    void employeeHasBeenAdded(Employee*);
    /** @brief employeeHasBeenDeleted To run when an employee has been deleted */
    void employeeHasBeenDeleted(Employee*);
};
```

# La bibliothèque graphique Qt

## Company : la classe MainGui

---

- Et l'implémentation qui correspond :

```
MainGui::MainGui(QWidget *parent) :
    QMainWindow(parent)
{
    company_ = new Company();
    companyIsLocal_ = true;
    setup();
}
MainGui::MainGui(Company* company, QWidget *parent) :
    QMainWindow(parent)
{
    company_ = company;
    companyIsLocal_ = false;
    setup();
}
MainGui::~MainGui() {
    if (companyIsLocal_) {
        delete company_;
    }
    while (!added_.empty()) {
        delete added_.back();
        added_.pop_back();
    }
}
```

On crée un objet **Company** local, et on garde un booléen pour penser à delete cet objet à la fin!

Notre objet **Company** contient les employés par agrégation: les employés que l'on va créer par l'interface devront donc être détruits lors de la destruction de notre interface

# La bibliothèque graphique Qt

## Company : la classe MainGui

---

- **setMenu**, que l'on utilise pour définir le menu en haut de la fenêtre:

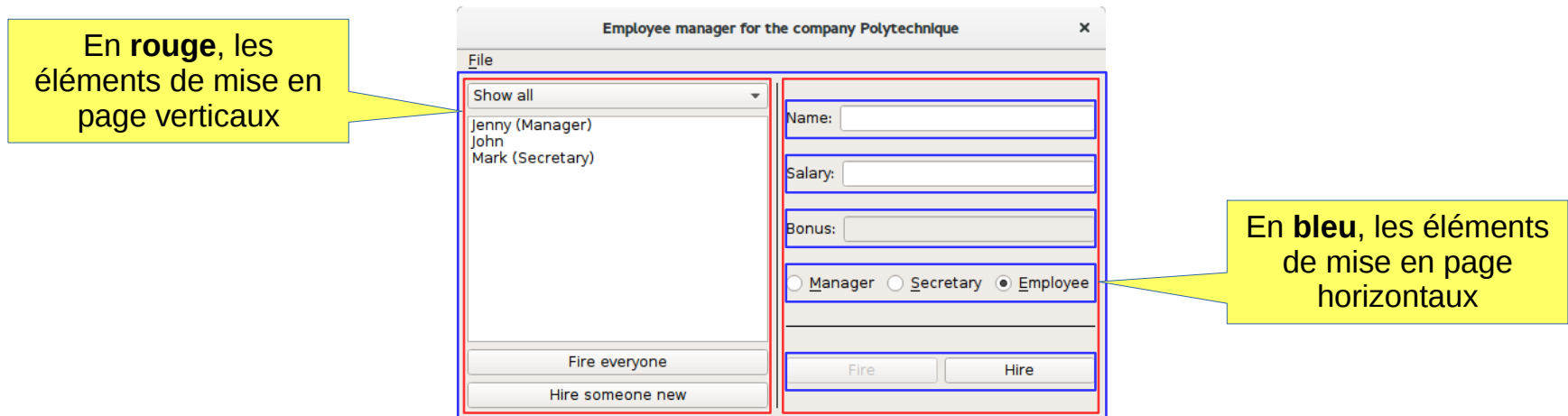
```
void MainGui::setMenu() {  
    // On crée un bouton 'Exit'  
    QAction* exit = new QAction(tr("E&xit"), this);  
    // On ajoute un raccourci clavier qui simulera l'appui sur ce bouton (Ctrl+Q)  
    exit->setShortcuts(QKeySequence::Quit);  
    // On connecte le clic sur ce bouton avec l'action de clore le programme  
    connect(exit, SIGNAL(triggered()), this, SLOT(close()));  
  
    // On crée un nouveau menu 'File'  
    QMenu* fileMenu = menuBar()->addMenu(tr("&File"));  
    // Dans lequel on ajoute notre bouton 'Exit'  
    fileMenu->addAction(exit);  
}
```

- Et **setUI** qui va nous permettre de définir la mise en page interne de notre fenêtre. Mais comment ça marche la mise en page ?

# La bibliothèque graphique Qt

## Company : mise en page

- Pour organiser une application, on a besoin d'utiliser des éléments de mise en page (**Layout**)
- Pour notre application, on utilisera principalement les verticaux et horizontaux (**QVBoxLayout** et **QHBoxLayout**)



- On cumulera ces éléments en les plaçant les uns à l'intérieur des autres pour créer des mises en pages plus complexes.



# La bibliothèque graphique Qt

## Company : mise en page

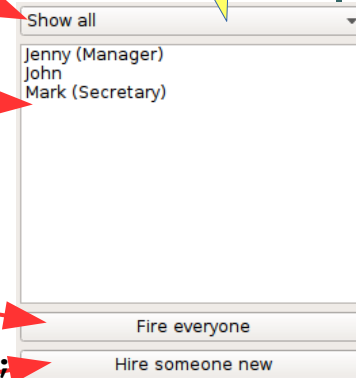
```
void MainGui::setUI() {
    // Le sélecteur pour filtrer ce que l'on souhaite dans la liste
    QComboBox* showCombobox = new QComboBox(this);
    showCombobox->addItem("Show all"); // Index 0
    showCombobox->addItem("Show only managers"); // Index 1
    showCombobox->addItem("Show only secretaries"); // Index 2
    showCombobox->addItem("Show only other employees"); // Index 3
    connect(showCombobox, SIGNAL(currentIndexChanged(int)),
            this, SLOT(filterList(int)));

    // La liste des employés
    employeesList = new QListWidget(this);
    employeesList->setSortingEnabled(true);
    connect(employeesList, SIGNAL(itemClicked(QListWidgetItem*)),
            this, SLOT(selectEmployee(QListWidgetItem*)));

    // Le bouton pour congédier tout le monde
    QPushButton* fireEveryoneButton = new QPushButton(this);
    fireEveryoneButton->setText("Fire everyone");
    connect(fireEveryoneButton, SIGNAL(clicked()), this, SLOT(fireEveryone()));

    // Le bouton pour remettre à zéro la vue et créer un nouvel employé
    QPushButton* hireSomeoneButton = new QPushButton(this);
    hireSomeoneButton->setText("Hire someone new");
    connect(hireSomeoneButton, SIGNAL(clicked()), this, SLOT(cleanDisplay()));
}
/* ... */
```

Lorsqu'on change l'élément sélectionné, le slot **filterList** sera appelé avec pour paramètre le nouvel index



# La bibliothèque graphique Qt

## Company : mise en page

---

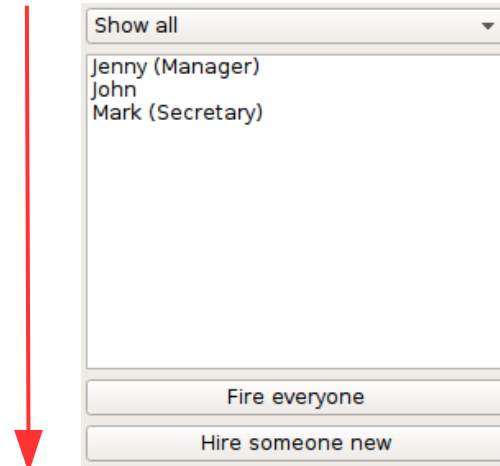
- Puis, on place nos éléments dans un **QVBoxLayout** qui va permettre de les organiser verticalement :

```
/* ... */
```

```
// Le premier layout, pour la colonne de gauche, dans lequel on insère les  
// éléments que l'on veut dans l'ordre dans lequel on veut qu'ils apparaissent
```

```
QVBoxLayout* listLayout = new QVBoxLayout;  
listLayout->addWidget(showCombobox);  
listLayout->addWidget(employeesList);  
listLayout->addWidget(fireEveryoneButton);  
listLayout->addWidget(hireSomeoneButton);
```

```
/* ... */
```



# La bibliothèque graphique Qt

## Company : mise en page

- On s'occupe maintenant des champs pour entrer les données, que l'on doit organiser dans leurs propres **QHBoxLayout** :

```
/* ... */
```

```
// Champ pour le nom
```

```
QLabel* nameLabel = new QLabel;  
nameLabel->setText("Name:");  
nameEditor = new QLineEdit;
```



```
QHBoxLayout* nameLayout = new QHBoxLayout;  
nameLayout->addWidget(nameLabel);  
nameLayout->addWidget(nameEditor);
```

```
// Champ pour le salaire
```

```
QLabel* salaryLabel = new QLabel;  
salaryLabel->setText("Salary:");  
salaryEditor = new QLineEdit;
```

```
QHBoxLayout* salaryLayout = new QHBoxLayout;  
salaryLayout->addWidget(salaryLabel);  
salaryLayout->addWidget(salaryEditor);
```

```
// Champ pour le bonus
```

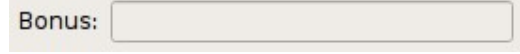
```
QLabel* bonusLabel = new QLabel;  
bonusLabel->setText("Bonus:");  
bonusEditor = new QLineEdit;  
bonusEditor->setDisabled(true);
```

```
QHBoxLayout* bonusLayout = new QHBoxLayout;  
bonusLayout->addWidget(bonusLabel);  
bonusLayout->addWidget(bonusEditor);
```

```
/* ... */
```

Désactivé par défaut car le bonus n'est valable que pour un Manager!

**QHBoxLayout:**  
Remplissage de gauche à droite



Bonus:

# La bibliothèque graphique Qt

## Company : mise en page

- On prépare ensuite le groupe de boutons radio pour choisir le type d'employé que l'on veut ajouter :

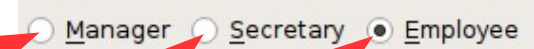
```
QRadioButton* managerRButton = new QRadioButton("&Manager", this);  
employeeTypeRadioButtons.push_back(managerRButton);
```

```
QRadioButton* secretaryRButton = new QRadioButton("&Secretary", this);  
employeeTypeRadioButtons.push_back(secretaryRButton);
```

```
QRadioButton* employeeRButton = new QRadioButton("&Employee", this);  
employeeRButton->setChecked(true);  
employeeTypeRadioButtons.push_back(employeeRButton);
```

```
QButtonGroup* employeeTypeButtonGroup = new QButtonGroup;  
employeeTypeButtonGroup->addButton(managerRButton);  
employeeTypeButtonGroup->addButton(secretaryRButton);  
employeeTypeButtonGroup->addButton(employeeRButton);  
connect(employeeTypeButtonGroup, SIGNAL(buttonClicked(int)),  
        this, SLOT(changedType(int)));
```

```
QHBoxLayout* employeeTypeLayout = new QHBoxLayout;  
employeeTypeLayout->addWidget(managerRButton);  
employeeTypeLayout->addWidget(secretaryRButton);  
employeeTypeLayout->addWidget(employeeRButton);
```



On crée un groupe de boutons qui permet de décocher automatiquement les autres boutons du groupe lorsqu'on en sélectionne un.

C'est aussi ce groupe qui émettra un signal lorsque le type d'employé sera changé: on le connecte à notre slot **changedType**

On place aussi les boutons dans un QHBoxLayout pour les organiser

# La bibliothèque graphique Qt

## Company : mise en page

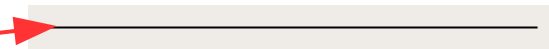
- On s'occupe enfin des boutons pour congédier/embaucher un employé :

```
// Trait horizontal de séparation
QFrame* horizontalFrameLine = new QFrame;
horizontalFrameLine->setFrameShape(QFrame::HLine);
```

```
// Bouton pour congédier la ou les personne(s)
// sélectionnée(s) dans la liste
fireButton = new QPushButton(this);
fireButton->setText("Fire");
fireButton->setDisabled(true);
connect(fireButton, SIGNAL(clicked()),
        this, SLOT(fireSelected()));
```

```
// Bouton pour embaucher la personne dont on
// vient d'entrer les informations
hireButton = new QPushButton(this);
hireButton->setText("Hire");
connect(hireButton, SIGNAL(clicked()),
        this, SLOT(createEmployee()));
```

```
// Organisation horizontale des boutons
QHBoxLayout* fireHireLayout = new QHBoxLayout;
fireHireLayout->addWidget(fireButton);
fireHireLayout->addWidget(hireButton);
```



Lorsqu'on clique sur ce bouton, le slot **fireSelected** sera appelé

Lorsqu'on clique sur ce bouton, le slot **createEmployee** sera appelé

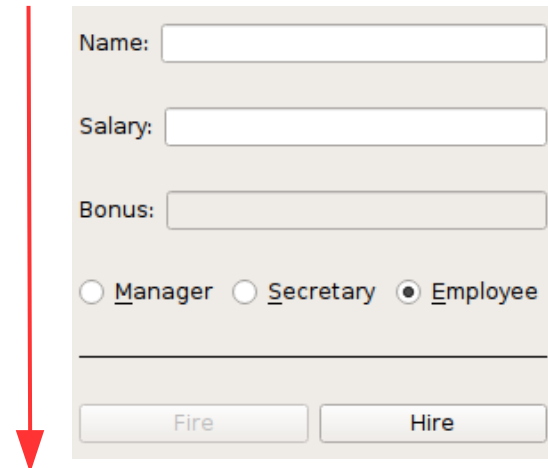
# La bibliothèque graphique Qt

## Company : mise en page

---

- On peut finalement placer les éléments de notre colonne de droite dans un **QVBoxLayout** pour les organiser

```
// Organisation pour la colonne de droite
QVBoxLayout* displayLayout = new QVBoxLayout;
displayLayout->addLayout(nameLayout);
displayLayout->addLayout(salaryLayout);
displayLayout->addLayout(bonusLayout);
displayLayout->addLayout(employeeTypeLayout);
displayLayout->addWidget(horizontalFrameLine);
displayLayout->addLayout(fireHireLayout);
```



# La bibliothèque graphique Qt

## Company : mise en page

- Enfin, on s'occupe de l'organisation de la fenêtre principale en y plaçant nos deux **Layout** précédemment préparés :

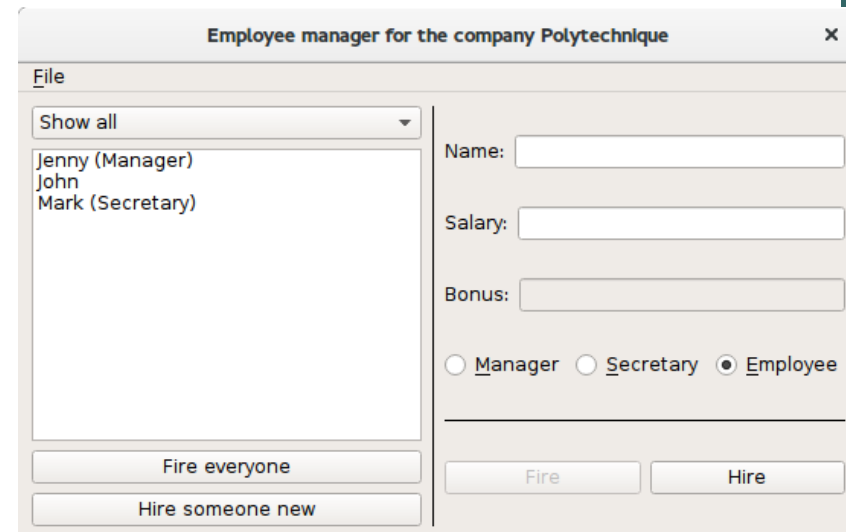
```
// Trait vertical de séparation
QFrame* verticalFrameLine = new QFrame;
verticalFrameLine->setFrameShape(QFrame::VLine);
```

```
// Organisation horizontale
QHBoxLayout* mainLayout = new QHBoxLayout;
mainLayout->addLayout(listLayout);
mainLayout->addWidget(verticalFrameLine);
mainLayout->addLayout(displayLayout);
```

```
// On crée un nouveau Widget, et on définit son
// layout pour celui que l'on vient de créer
QWidget* widget = new QWidget;
widget->setLayout(mainLayout);
```

```
// Puis on définit ce widget comme le widget
// centrale de notre classe
setCentralWidget(widget);
```

```
// Enfin, on met à jour le titre de la fenêtre
string title = "Employee manager for the company " + company_>getName();
setWindowTitle(title.c_str());
```



# La bibliothèque graphique Qt

## Company : mise en page

---

- On fera appel à **setMenu** et à **setUI** dans la méthode **setup** à laquelle on fait appel dans nos constructeurs :

```
void MainGui::setup() {
    currentFilterIndex_ = 0;

    // On créé notre menu et notre UI
    setMenu();
    setUI();

    // On connecte les signaux de notre company aux
    // slots créés localement pour agir suite à ces signaux
    connect(company_, SIGNAL(employeeAdded(Employee*)),
            this, SLOT(employeeHasBeenAdded(Employee*)));
    connect(company_, SIGNAL(employeeDeleted(Employee*)),
            this, SLOT(employeeHasBeenDeleted(Employee*)));

    // On charge la liste des employés actuels
    loadEmployees();
}
```



# La bibliothèque graphique Qt

## Company : slots

- Il nous faut maintenant un moyen de charger les employés dans notre liste graphique depuis le modèle :

```
void MainGui::loadEmployees() {  
    // On s'assure que la liste est vide  
    employeesList->clear();  
    // Puis, pour tous les employés dans Company  
    int max = company_->getNumberEmployees();  
    for (int i = 0; i < max; i++) {  
        // On récupère le pointeur vers l'employé  
        Employee* employee = company_->getEmployee(i);  
        if (employee == nullptr) {  
            continue;  
        }  
        // Et on l'ajoute en tant qu'item de la liste:  
        // le nom sera affiché, et le pointeur sera contenu  
        QListWidgetItem* item = new QListWidgetItem(  
            QString::fromStdString(employee->getName()), employeesList);  
        item->setData(Qt::UserRole, QVariant::fromValue<Employee*>(employee));  
        item->setHidden(filterHide(employee));  
    }  
}
```

**QString::fromStdString** permet de transformer un **string** en **QString** pour l'utiliser dans l'interface

**QVariant::fromValue<T>** permet de convertir un élément de type **T** en objet **QVariant**, qui peut être stocké en tant que contenu d'un item d'une liste par exemple

# La bibliothèque graphique Qt

## Company : slots

- Le slot **filterList** permettant de filtrer notre liste lorsqu'on change le type d'employé que l'on souhaite afficher :

```
bool MainGui::filterHide(Employee* employee) {
    switch (currentFilterIndex_) {
    case 1:
        return (typeid(*employee) != typeid(Manager));
    case 2:
        return (typeid(*employee) != typeid(Secretary));
    case 3:
        return (typeid(*employee) != typeid(Employee));
    case 0:
    default:
        return false;
    }
}

void MainGui::filterList(int index) {
    currentFilterIndex_ = index;
    for (int i = 0; i < employeesList->count(); ++i) {
        QListWidgetItem *item = employeesList->item(i);
        Employee* employee = item->data(Qt::UserRole).value<Employee*>();
        item->setHidden(filterHide(employee));
    }
}
```

On récupère le pointeur d'Employee contenu dans notre item de liste afin de le comparer

# La bibliothèque graphique Qt

## Company : slots

- Le slot **selectEmployee** permettant de mettre dans la colonne de droite les informations de l'employé sélectionné :

```
void MainGui::selectEmployee(QListWidgetItem* item) {
    Employee* employee = item->data(Qt::UserRole).value<Employee*>();

    nameEditor->setDisabled(true);
    nameEditor->setText(QString::fromStdString(employee->getName()));

    salaryEditor->setDisabled(true);
    salaryEditor->setText(QString::number(employee->getSalary()));

    bonusEditor->setDisabled(true);
    if (typeid(*employee) == typeid(Manager)) {
        bonusEditor->setText(
            QString("%1\\% (included in salary)").arg(((Manager*)employee)->getBonus()));
    } else {
        bonusEditor->setText("");
    }

    /* ... */
}
```

Les champs sont désactivés pour la modification, on veut juste afficher les informations ici

Pour un Manager, on affiche le pourcentage, mais le montant est déjà inclus dans le champs salaire

# La bibliothèque graphique Qt

## Company : slots

- Le slot **selectEmployee** permettant de mettre dans la colonne de droite les informations de l'employé sélectionné :

```
/* ... */
```

```
list<QRadioButton*>::iterator end = employeeTypeRadioButtons.end();  
for (auto it = employeeTypeRadioButtons.begin(); it != end; it++) {  
    (*it)->setDisabled(true);
```

```
    bool checked = false;
```

```
    if ((typeid(*employee) == typeid(Manager) && (*it)->text().endsWith("Manager")) ||  
        (typeid(*employee) == typeid(Secretary) && (*it)->text().endsWith("Secretary")) ||  
        (typeid(*employee) == typeid(Employee) && (*it)->text().endsWith("Employee"))) {  
        checked = true;
```

```
    }
```

```
    (*it)->setChecked(checked);
```

```
}
```

```
fireButton->setDisabled(false);  
hireButton->setDisabled(true);
```

```
}
```

On parcourt tous les boutons radio, on ne veut checker que celui qui correspond

Si le type de notre **Employee** correspond au bouton, on indique donc qu'on doit le checker !

On peut aussi désactiver le bouton pour **embaucher**, et activer le bouton pour **congédier**

# La bibliothèque graphique Qt

## Company : slots

- Le slot **cleanDisplay** permettant de restaurer la vue initiale :

```
void MainGui::cleanDisplay() {  
    nameEditor->setDisabled(false);  
    nameEditor->setText("");  
  
    salaryEditor->setDisabled(false);  
    salaryEditor->setText("");  
  
    bonusEditor->setDisabled(true);  
    bonusEditor->setText("");  
  
    list<QRadioButton*>::iterator end = employeeTypeRadioButtons.end();  
    for (auto it = employeeTypeRadioButtons.begin(); it != end; it++) {  
        (*it)->setDisabled(false);  
        if ((*it)->text().endsWith("Employee")) {  
            (*it)->setChecked(true);  
        }  
    }  
  
    employeesList->clearSelection();  
    fireButton->setDisabled(true);  
    hireButton->setDisabled(false);  
  
    nameEditor->setFocus();  
}
```

On vide les champs texte, et on les réactive pour l'écriture

On parcourt les boutons radio pour sélectionner celui pour créer un employé simple

On désélectionne tout ce qui était sélectionné dans la liste des employés

On active le bouton pour embaucher, et on désactive le bouton pour congédier

On met le curseur dans le champs pour éditer le nom, afin que l'utilisateur puisse directement écrire!

# La bibliothèque graphique Qt

## Company : slots

- Le slot **changedType** lorsqu'on sélectionne un type d'employé :

```
void MainGui::changedType(int index) {  
    if (index == -2) {  
        bonusEditor->setDisabled(false);  
    } else {  
        bonusEditor->setDisabled(true);  
    }  
}
```

On veut n'activer le champs pour entrer le bonus que si c'est un **Manager** qui est sélectionné dans les boutons radio. L'index **-2** correspond à cette situation dans notre cas!

- Le slot **fireSelected** lorsqu'on clique sur le bouton correspondant :

```
void MainGui::fireSelected() {  
    vector<Employee*> toDelete;  
    for (QListWidgetItem *item : employeesList->selectedItems()) {  
        toDelete.push_back(item->data(Qt::UserRole).value<Employee*>());  
    }  
  
    for (Employee* e : toDelete) {  
        company_->delEmployee(e);  
    }  
}
```

On fait appel à **delEmployee** de **Company** pour chaque élément de notre **QListWidget**. On les a mis dans un vecteur au préalable car la suppression des employés va modifier la **QlistWidget**. On fait donc cette suppression en deux étapes.

# La bibliothèque graphique Qt

## Company : slots

- Le slot **fireEveryone** est identique à **fireSelected**, excepté pour le fait que l'on place dans le vecteur tous les items de la liste:

```
void MainGui::fireEveryone() {  
    vector<Employee*> toDelete;  
    for (int i = 0; i < employeesList->count(); ++i) {  
        QListWidgetItem *item = employeesList->item(i);  
        toDelete.push_back(item->data(Qt::UserRole).value<Employee*>());  
    }  
  
    for (Employee* e : toDelete) {  
        company_->delEmployee(e);  
    }  
}
```

L'écriture de la boucle **for** est différente car on ne peut pas itérer directement sur un **QListWidget**



# La bibliothèque graphique Qt

## Company : slots

---

- On veut aussi créer un nouvel employé lorsqu'on clique sur **Hire**, le slot **createEmployee** est là pour ça :

```
void MainGui::createEmployee() {  
    // On va créer un nouvel employé que l'on placera dans ce pointeur  
    Employee* newEmployee;  
  
    // On crée le bon type d'employé selon le cas  
    QRadioButton* selectedType = 0;  
    list<QRadioButton*>::iterator end = employeeTypeRadioButtons.end();  
    for (auto it = employeeTypeRadioButtons.begin(); it != end; it++) {  
        if ((*it)->isChecked()) {  
            selectedType = *it;  
            break;  
        }  
    }  
  
    /* ... */  
}
```

On passe les boutons radio un par un jusqu'à rencontrer celui qui est coché. Dès qu'on l'a trouvé, on saura quel type d'employé on cherche à créer!



# La bibliothèque graphique Qt

## Company : slots

- On veut aussi créer un nouvel employé lorsqu'on clique sur **Hire**, le slot **createEmployee** est là pour ça :

```
/* ... */

// On crée le bon type d'employé selon le cas
if (selectedType->text().endsWith("Manager")) {
    newEmployee = new Manager(nameEditor->text().toString(),
                              salaryEditor->text().toDouble(),
                              bonusEditor->text().toDouble());
} else if (selectedType->text().endsWith("Secretary")) {
    newEmployee = new Secretary(nameEditor->text().toString(),
                                salaryEditor->text().toDouble());
} else {
    newEmployee = new Employee(nameEditor->text().toString(),
                               salaryEditor->text().toDouble());
}

// On ajoute le nouvel employé créé à la company
company->addEmployee(newEmployee);
// Mais on le stocke aussi localement pour pouvoir le supprimer plus tard
added_.push_back(newEmployee);
}
```

**toString()** et **toDouble()** permettent respectivement de passer d'un **QString** à un string et un double

On appelle la méthode **addEmployee** de **Company** pour ajouter notre employé nouvellement créé

# La bibliothèque graphique Qt

## Company : slots

---

- Enfin, on veut réagir lorsqu'un employé est ajouté dans la **Company** sur laquelle on travaille :

```
void MainGui::employeeHasBeenAdded(Employee* employee) {  
    // On ajoute le nouvel employé comme item de la QListWidget  
    QListWidgetItem* item = new QListWidgetItem(  
        QString::fromStdString(employee->getName()), employeesList);  
    item->setData(Qt::UserRole, QVariant::fromValue<Employee*>(employee));  
  
    // On change la visibilité de notre nouvel employé selon  
    // le filtre actuel  
    item->setHidden(filterHide(employee));  
}
```

# La bibliothèque graphique Qt

## Company : slots

- ... et lorsqu'un employé est supprimé de la **Company** sur laquelle on travaille :

```
void MainGui::employeeHasBeenDeleted(Employee* e) {  
    // On cherche dans notre QListWidget l'employé pour lequel le  
    // signal a été envoyé, afin de l'en retirer  
    for (int i = 0; i < employeesList->count(); ++i) {  
        QListWidgetItem *item = employeesList->item(i);  
        Employee* employee = item->data(Qt::UserRole).value<Employee*>();  
        if (employee == e) {  
            // delete sur un QListWidget item va automatiquement le retirer de la liste  
            delete item;  
            // Si l'employé faisait partie de ceux créés localement, on veut le supprimer.  
            auto it = std::find(added_.begin(), added_.end(), e);  
            if (it != added_.end()) {  
                delete *it;  
                added_.erase(it);  
            }  
            // L'employé ne devrait être qu'une fois dans la liste...  
            break;  
        }  
    }  
    // On remet à zéro l'affichage de la colonne de gauche étant  
    // donné que les employés sélectionnés ont été supprimés  
    cleanDisplay();  
}
```

# La bibliothèque graphique Qt

## Company : interface finale

Employee manager for the company Polytechnique

File

Show all

- Jenny (Manager)
- John
- Mark (Secretary)

Name: John

Salary: 15000

Bonus:

☐ Manager ☐ Secretary ☒ Employee

Fire everyone

Hire someone new

Fire Hire

Employee manager for the company Polytechnique

File

Show all

- Jenny (Manager)
- John
- Mark (Secretary)

Name: Jenny (Manager)

Salary: 13800

Bonus: 15% (included in salary)

☒ Manager ☐ Secretary ☐ Employee

Fire everyone

Hire someone new

Fire Hire

Employee manager for the company Polytechnique

File

Show all

- Jenny (Manager)
- John
- Mark (Secretary)

Name: Mark (Secretary)

Salary: 16000

Bonus:

☐ Manager ☒ Secretary ☐ Employee

Fire everyone

Hire someone new

Fire Hire

Employee manager for the company Polytechnique

File

Show all

- Jenny (Manager)
- John
- Mark (Secretary)

Name: Gédéon

Salary: 60000

Bonus: 30

☒ Manager ☐ Secretary ☐ Employee

Fire everyone

Hire someone new

Fire Hire