# Boîtes à outils pour la construction d'interfaces usager graphiques



ÉCOLE POLYTECHNIQUE MONTRÉAL



## Les boîtes à outils

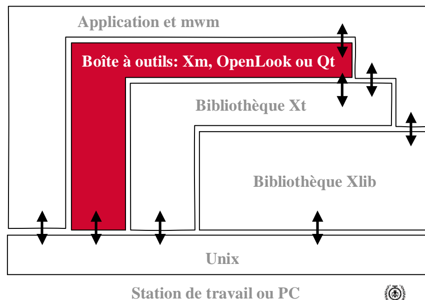
- Extension aux systèmes de fenêtrage de base
  - Bibliothèque de composantes dédiées à la construction d'interfaces
  - Construite comme une couche supplémentaire au-dessus du système de fenêtrage de base
  - Prend en charge une grande partie de la gestion des événements
- Pourquoi des boîtes à outils?
  - on travaille avec un langage de plus haut niveau alors la conception est plus rapide et aisée
  - on retrouve souvent les mêmes éléments dans une interface
  - on produit des interfaces plus standards



ÉCOLE POLYTECHNIQUE MONTRÉAL



## Architecture – X11/Xt




Station de travail ou PC



ÉCOLE POLYTECHNIQUE MONTRÉAL

## Les boîtes à outils

- Certaines boîtes à outils sont portables
  - Motif
  - OpenLook
  - Qt
  - Java Swing
- D'autres sont spécifiques à un système d'exploitation
  - Windows 95/98/NT
  - Presentation Manager (OS2)
  - MacApp





## Les boîtes à outils

La boîte à outils détermine l'apparence et le comportement *("look and feel")*

- Apparence
  - Boutons
  - Disposition des menus
  - Décorations des fenêtres
- Comportement
  - Règles d'allocation de la souris et du clavier
  - Assignation des touches accélératrices
- Ce sont les V et C de l'architecture MVC





## Composantes

Les composantes d'une boîte à outils sont appelés ***Widgets*** ("window gadget")

- Un Widget c'est:
  - Une instance d'une classe dérivée d'une fenêtre
  - Un objet (donc fonctions et données) interactif qui représente un élément courant d 'interface usager graphique comme p.e. bouton poussoir, barre de défilement, menu…
  - Une fenêtre possédant une apparence et un comportement bien déterminés
  - Une "brique" réutilisable et configurable servant à la construction de la partie interface d'une application
  - le nom de baptême donné par Xt (« X toolkit ») mais on l'appelle aussi *contrôle* (MS Windows, Mac)

## Composantes: types courants

- bouton poussoir
  - manipule une variable discrète ou déclenche une fonctionnalité
  - modèle
    - variable à deux ou à un petit nombre (boutons radio) d'états possibles
- barre de défilement ou bouton glissoir
  - manipule une variable continue bornée
  - modèle
    - valeur courante et deux valeurs extrêmes
    - [taille de la fenêtre visible] : si la longueur du signet indique la proportion de l'information qui est visible simultanément
    - [taille d'une page] : si le saut de page est offert





## Composantes: types courants

- menu (défilant, « popup »)
  - conceptuellement assez semblable au bouton poussoir
    - manipule une variable discrète ou déclenche une fonctionnalité
  - plus commode pour offrir un grand nombre de choix
- boîte de texte
  - permet d'entrer ou de modifier du texte (qui représente peut-être un nombre)
  - modèle
    - chaîne de caractères





## Composantes: types courants

- dialogue
  - Composante autonome apparaissant dans une fenêtre séparée
  - Généralement composé de plusieurs composantes atomiques
  - Fournit une interface spécifiquement construite en fonction des besoins de l'application.
  - Deux modes d'interaction:
    - Suspend l'application jusqu'à sa complétion (modal)
    - Fonctionne de façon parallèle à l'application (non-modal)
  - Exemples de dialogues courants:
    - Boîte de message
    - Boîte de sélection d'un fichier
    - Boîte de contrôle d'impression
    - Boîte à onglets

## Organisation hiérarchique

- Une boîte à outils présente les Widgets sous forme d'une hiérarchie de classes







## Organisation hiérarchique

Une application utilise les Widgets de la boîte à outils en construisant une hiérarchie d'instances.





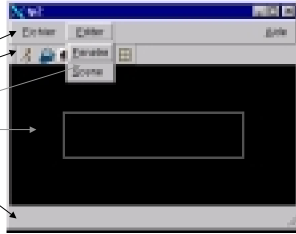


## Organisation hiérarchique

Les Widgets sont construits à l'intérieur d'une hiérarchie de fenêtres

## Exemple d’application construite avec Qt

- Fenêtre principale de l’application construite à l’aide de plusieurs composantes, dont:
  - La barre de menu
  - La barre d’outils
  - Des menus défilants
  - La zone graphique
  - La zone de statut

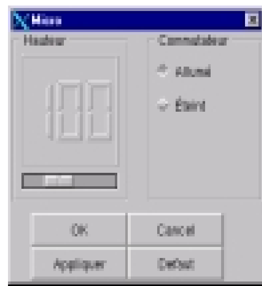






## Disposition

- Certaines composantes comme les dialogues contiennent plusieurs autres composantes
- La disposition de chaque composante à l’intérieur peut être déterminée:
  - de façon absolue
  - de façon relative à l’aide de contraintes géométriques entre composantes de taille fixe
  - comme le précédent mais avec en plus des contraintes géométriques pour chaque composante, de taille variable
  - ...



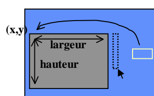





## Disposition absolue

- Méthode simple et rapide de conception des composantes à l’intérieur d’une fenêtre de dialogue
- Potentiellement très visuelle (« WYSIWYG »)
- Réagit plus ou moins bien à la modification de la taille de la fenêtre-mère du contenant (p.e dialogue)

  solution: écrire du code application recalculant la position et la taille des composantes, qui sera exécuté suite à un événement de modification de la taille de la fenêtre-mère



## Disposition relative: traverses et ressorts

- La position d'une composante peut être définie par rapport à la position d'une autre composante (à gauche, à droite, en haut ou en bas) ou par rapport au bord de la fenêtre qui la contient (fixée à la bordure gauche, droite, du haut ou du bas) sous forme de *contraintes géométriques* entre les composantes

- Automatise le positionnement et dimensionnement des composantes suite à une modification de la taille de la fenêtre-mère
- La spécification de contraintes ne fait pas très « WYSIWYG »
  solution: métaphore des *traverses et ressorts*

- La taille intrinsèque d'une composante primitive est généralement fixée ou au mieux indépendante de sa nature (avec un ressort dedans)





## Disposition relative: compression et étirement

- En plus de contraintes géométriques entre les composantes, on fournit des informations quant à la taille (hauteur, largeur) minimale/maximale/idéale de chaque composante, indiquant jusqu'à quel point on peut les comprimer ou les étirer

- La taille finale d'une composante peut varier en fonction de sa nature (p.e. la longueur de l'étiquette pour un bouton poussoir) ou du contexte (p.e. les dimensions de la fenêtre-mère)







## Disposition -- *Qt*

- Absolue
  - tous les widgets de la bibliothèque Qt peuvent être positionnés et dimensionnés à l'aide de la méthode `setGeometry()`
- Relative, compression et étirement
  - utilise des objets appelés *gestionnaires de géométrie* (« layout managers »)
    - `QHBoxLayout` et `QVBoxLayout` permettent de maintenir une relation horizontale ou verticale entre certaines composantes
    - `QGridLayout` permet de maintenir une relation matricielle entre certaines composantes
  - ces gestionnaires ont des méthodes pour insérer un espace de taille fixe (`addSpacing()`) ou variable (`addStretch()`) et pour geler la taille du contenant (`freeze()`))
  - on peut imbriquer des gestionnaires de géométrie
  - pour calculer des tailles selon la résolution de l'écran: `QApplication::desktop(){->width(), ->height()}`

## Disposition -- *Qt*
## exemple de disposion relative

```
MonDialogue::MonDialogue() {
// créer les composantes, en spécifiant des tailles minimales et maximales au besoin
QPushButton* b1 = new QPushButton(« bouton 1 », this);
b1->setMinimumSize( b1->sizeHint());
QPushButton* b2 = new QPushButton(« bouton 2 », this);
b2->setMinimumSize( b2->sizeHint());
// créer un (ou plusieurs) gestionnaire(s) de géométrie
QHBoxLayout* gest = new QHBoxLayout(this);
// inscrire chacune des composantes
gest->addWidget(b1);
gest->addWidget(b2);
// déclencher la gestion de la géométrie
gest->activate();
}
```

## Disposition -- *Java Swing*

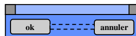
- La classe `Jcomponent` dont dérivent toutes les composantes définit les propriétés
  - `bounds, minimumSize, maximumSize, preferredSize`
- Absolue
  - tous les widgets de la bibliothèque Java Swing peuvent être positionnés et dimensionnés à l'aide de la méthode `setBounds()`
- Relative
  - utilise elle aussi des gestionnaires de géométrie
    - p.e. `BoxLayout(widget, {X_AXIS, Y_AXIS})`
  - plusieurs types de contenants existent, avec leurs routines de création
    - p.e. `createHorizontalBox(), createVerticalBox()`
  - ces contenants ont des méthodes pour insérer un espace de taille fixe (`createHorizontalStrut(), createVerticalStrut()`) ou variable (`createHorizontalGlue(), createVerticalGlue()`)





## Disposition -- *Java Swing*
## exemple de disposion relative

```
// créer un contenant
Box box = Box.createHorizontalBox();
// créer les composantes, en spécifiant des tailles minimales, maximales ou préférées au besoin
Jbutton b1 = new Jbutton(« ok »);
b1.setPreferredSize(new Dimension(50,10));
Jbutton b2 = new Jbutton(« annuler »);
// inscrire chacune des composantes
box.add(b1);
box.add(Box.createHorizontalStrut(50));
box.add(b2);
// associer un gestionnaire de géométrie
box.setLayout(new BoxLayout(box,BoxLayout.X_AXIS));
```



## Configuration de composantes

- Certains attributs d 'un widget sont modifiables par le concepteur d 'application ou l 'usager (p.e. couleur d 'avant- et d 'arrière-plan, police, étiquette, taille de la bordure…) alors que d 'autres ne le sont pas (p.e forme arrondie d 'un bouton ou toute apparence spécifique à une boîte à outils donnée)
- Cette possibilité de configuration permet la conception de composantes plus génériques et offre beaucoup de flexibilité
- Comment configurer un widget?
  - par des appels de fonction, lors de la création ou la modification d 'un widget
  - par un mécanisme de préférences modifiables par l 'usager à travers l 'interface de l 'application
  - en spécifiant certains paramètres optionnels à la ligne de commande
  - en modifiant un fichier de ressources pour l 'application





## Configuration de composantes

- une *ressource* de widget est une donnée contenant de l 'information sur son interface (surtout son apparence mais aussi son comportement)
- sa dissociation du code permet
  - l 'utilisation d 'un éditeur de ressources
  - de rendre la configuration accessible aux concepteurs visuels d 'interfaces
  - de modifier la configuration sans pour autant devoir recompiler l 'application
- la syntaxe de spécification de ressources permet souvent d 'indiquer une ressource commune à un groupe de widgets (p.e. une valeur par défaut)
- Comment garder le code et ses ressources ensemble?
  - MacOS: tout fichier comprend un « data fork » et un « resource fork »
  - MSWindows: un compilateur de ressources les assemble puis les ajoute à l 'exécutable
  - X: définit des chemins de recherche standard pour trouver les ressources





## Configuration de composantes: comparaison

- dans le code
  - garantit la valeur d 'une ressource critique
  - toute modification requiert recompilation
- à l 'interface
  - contrôle sur la flexibilité offerte
  - à recommencer à chaque exécution (ou sinon c 'est qu 'on utilise un fichier)
- à la ligne de commande
  - flexibilité et facilité d 'utilisation
  - laborieux si on a un grand nombre de ressources à fixer
- dans un fichier
  - flexibilité et personnalisation d 'une application
  - rapidité de modification
  - une mauvaise configuration peut rendre l 'application inopérante ou dangereuse
  - les erreurs de syntaxe ne sont pas rapportées

## Configuration -- *Qt*

- pas grand chose hormis la spécification directe dans le code
- méthode `Qstring QObject::tr(Qstring s)`
  - traduit s à partir d 'une `QMessageFile`
  - pour internationaliser une interface
- méthode `Qapplication::setStyle({MotifStyle,WindowsStyle})`
  - pour donner un style global bien reconnu à une application
  - on peut aussi spécifier ceci à la ligne de commande, p.e `-style windows`





## Configuration -- *Java Swing*

- spécification de propriétés individuelles dans le code
  - p.e. `bouton.setBackground(Color.black);`
- spécification de propriétés pour un groupe de widgets
  - p.e. `UIManager.put(« Button.background », Color.black);`

**UIResources**





## Configuration -- *X/Motif*

- dans le code ou à la ligne de commande
- dans un fichier de ressources
  - syntaxe: <widget(s)>.<ressource>:<valeur>
  - où <widget(s)> est un chemin dans la hiérarchie d 'instances, utilisant des noms d 'objets ou de classes séparés par des points, et possiblement des étoiles pouvant représenter n 'importe quoi

```
appli.colonneBoutons.boutonOk.foreground: blue
*foreground: red
appli*PushButton.foreground: green
```

  - emplacements standards de fichiers de ressources pour « appli »
    - ~/Appli
    - /usr/lib/X11/app-defaults/Appli
    - ~/.Xdefaults

## Modes de rétro-action des composantes

Les composantes doivent être capable de signaler à l'application les événements qui se produisent au niveau de l'interface. Plusieurs méthodes de rétroaction sont envisageables:

- Les gestionnaires d'événements
- Les fonctions de rappel
- Les fonctions d'action
- Les fonctions virtuelles surchargées
- Les macros de connexion événements-méthodes
- Les signaux et prises (« signals and slots »)





## Rétro-action: le gestionnaire d'événements

- Une fonction ou une méthode associée à un widget, acceptant directement tous les événements détectés par le système de fenêtrage quel que soit leur type
- Un seul point de traitement pour tous les types d'événement
- Remet tout le travail entre les mains du programmeur de l'application





## Rétro-action: les fonctions de rappel

- Fonctions séparées des classes et possédant une interface standard
- Un widget contient, dans sa définition, un certain nombre de conditions auxquelles le programmeur peut associer des fonctions de rappel, par exemple:
  - Le widget vient d'être activé (ActivateCallback pour un Pushbutton)
  - L'état du widget vient d'être basculé (ToggleButton)
  - La valeur du widget vient de changer (ValueChangedCallback pour un Scale)
- Une fonction de rappel est donc appelée en réponse à un événement *prévu et prédéfini* pour un widget
- Les fonctions de rappel n'ont pas accès au modèle de données car elles sont soit séparées des classes, soit statiques
- Les fonctions de rappel doivent avoir une signature fixe mais le compilateur n'est pas en mesure de vérifier si c'est bien le cas.

## Rétro-action: les fonctions d'action

- Fonctions séparées des classes et possédant une interface standard
- Les fonctions d'action sont directement associées à des événements de bas-niveau du système de fenêtrage au travers d'une table locale pour chaque widget, par exemple:

| **événement** | **fonction** |
|---|---|
| ❒ Boutton1 pesé | → appeler la fonction fctActionEntrerObjet |
| ❒ Boutton2 pesé | → appeler la fonction fctActionEffacerObjet |
| ❒ Expose | → appeler la fonction fctActionRedessiner |

- Une fonction d'action est donc appelée en réponse à un événement *non prévu* pour un widget
- Mêmes problèmes d'accès aux données et de signature que pour les fonctions de rappel





## Rétro-action: les fonctions virtuelles surchargées

- Tous les widgets dérivent d'une classe commune définissant toutes les méthodes virtuelles de traitement des événements
- Par exemple dans Qt, la classe QWidget définit les méthodes suivantes:
  - virtual void mousePressEvent ( QMouseEvent * )
  - virtual void mouseReleaseEvent ( QMouseEvent * )
  - virtual void mouseDoubleClickEvent ( QMouseEvent * )
  - virtual void mouseMoveEvent ( QMouseEvent * )
  - virtual void keyPressEvent ( QKeyEvent * )
  - virtual void keyReleaseEvent ( QKeyEvent * )
  - virtual void focusInEvent ( QFocusEvent * )
  - virtual void focusOutEvent ( QFocusEvent * )
  - virtual void enterEvent ( QEvent * )
  - virtual void leaveEvent ( QEvent * )
  - virtual void paintEvent ( QPaintEvent * )
  - virtual void moveEvent ( QMoveEvent * )
  - virtual void resizeEvent ( QResizeEvent * )
  - virtual void closeEvent ( QCloseEvent * )





## Rétro-action: les fonctions virtuelles surchargées

- Méthode très « orientée-objet »
- Permet un accès efficace au modèle de données (les fonctions sont membres de la sous-classe du widget choisi)
- Ce mécanisme, lorsqu'il est utilisé pour traiter tous les événements dans une application, exige, par exemple, de sous-classer chaque bouton pour traiter l'événement ButtonPress → problème de mise en application à très grande échelle

## Rétro-action: les macros de connexion

- Méthode utilisée par les MFC pour connecter un événement (message) à une méthode d'une classe
- Méthode très peu « orientée-objet »
- Utilise le préprocesseur de C pour générer les connexion
- Difficile à lire et à comprendre
- Nécessite l'utilisation d'un outil d'aide à la conception pour la construction de l'interface





## Rétro-action: les signaux et les prises

- Mécanisme propre à Qt pour établir la connexion entre:
  - une condition déterminée par le programmeur de l'application et
  - une méthode quelconque d'un objet
- Les signaux et les prises sont définis au niveau de la classe
- Un signal définit la signature de la méthode qui pourra être appelée
- Un signal ne définit pas une méthode comme telle
- Une prise (« slot ») est une méthode normale d'une classe
- La signature de la prise doit correspondre exactement à la signature du signal sur lequel le programmeur veut la brancher





## Rétro-action: les signaux et les prises

- Chaque classe définissant un signal ou une prise doit être traitée à l'aide du compilateur « moc » afin de produire une spécification acceptable pour le compilateur C++
- Un signal peut être émis n'importe où dans une méthode de la classe qui déclare le signal à l'aide de la macro `emit`
- La connexion entre un signal et une prise est établie à l'aide de la méthode statique `connect`

```
QObject::connect(emetteur,SIGNAL(nom signal),
                 recepteur,SLOT(nom_prise) );
```

## Structure typique d 'une application -- *X/Motif*

- Fichiers de déclaration
  - `Xm<widget>.h` pour chacune des composantes utilisées
- Initialisation de la boîte à outils: `Xt{Va}AppInitialize()`
  - ouvrir la connexion au serveur X
  - bâtir une base de données de spécifications de ressources
  - créer la composante mère de l 'application
- Création des composantes de l 'application
  - créer chaque widget et initialiser ses ressources: `XmCreate<widget>()`
  - avertir un widget parent (contenant) qu 'il a la responsabilité de gérer la géométrie et l 'affichage d 'autres widgets: `XtManageChild()`
- Inscription des fonctions de rappel et d 'action
- Réalisation de la hiérarchie de widgets créés: `XtRealizeWidget()`
  - calculer les tailles et positions des différentes composantes
  - afficher leurs fenêtres
- Initialisation du modèle de données de l 'application
- Boucle de gestion d 'événements: `XtAppMainLoop()`





## Structure typique d 'une application -- *Qt*

- Fichiers de déclaration
  - `q<widget>.h` pour chacune des composantes utilisées
- Initialisation de la boîte à outils: `QApplication appli(argc, argv);`
  - ouvrir la connexion au serveur X
  - créer la composante mère de l 'application
- Création des composantes de l 'application
  - créer chaque widget et initialiser ses attributs: `new Q<widget>();`
  - brancher les événements au code de l 'application: `connect()`, virtuelles
  - gérer la géométrie des widgets primitifs s 'il s 'agit d 'un contenant: `addWidget()` … `activate()`
- Indiquer le widget principal de l 'application: `appli->setMainWidget();`
- Demander d 'afficher les fenêtres: `show()`
- Initialisation du modèle de données de l 'application
- Boucle de gestion d 'événements: `appli.exec()`





## Conception d'interfaces: quelques conseils

- Ne pas réinventer la roue!
  - bien connaître la boîte à outils
  - choisir la composante (« widget ») la plus proche de ce qu 'on veut faire
- Toujours permettre à l 'usager de faire marche arrière
- Fournir des explications sur la signification des icônes
- Respecter les conventions
  - p.e. bouton de droite pour « pop-up »; items de menu avec « … » ou « ▸ »
- Indiquer visuellement qu 'une fonction ne s 'applique pas pour l 'instant
  - p.e. item de menu affiché plus pâle
- Si une application utilise surtout le clavier, permettre de tout faire au clavier
- Faire apparaître un indicateur de « travail en cours » lorsqu 'une opération prend du temps à s 'effectuer
- Offrir des touches d 'accélération pour les usagers avertis (« power users »)