Une architecture pour améliorer la réutilisatibilté des interfaces graphiques

Eric Lecolinet

▶ **To cite this version:**

Eric Lecolinet. Une architecture pour améliorer la réutilisatibilté des interfaces graphiques. Annexes des actes de la 31e conférence francophone sur l'Interaction Homme-Machine (IHM 2019), Dec 2019, Grenoble, France. pp.13:1-8, 10.1145/3366551.3370352. hal-02388858

HAL Id: hal-02388858

https://hal.science/hal-02388858v1

Submitted on 2 Dec 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Un modèle architectural facilitant la réutilisabilité des interfaces graphiques

An architectural model for Improving the reusability of graphical user interfaces

Eric Lecolinet
LTCI, Télécom Paris, Institut Polytechnique de Paris
Paris, France
eric.lecolinet@telecom-paris.fr

ABSTRACT

Graphical interface programming remains a laborious and time-consuming exercise of the bulk of the interaction that usually has to be coded using a programming language. This code, often wordy and not very readable, impacts the reusability of the interfaces and the iterative design, even minor modifications of the user interface requiring it to be substantially modified. We propose an architectural model and an experimental toolkit that makes modifications easier.

CCS CONCEPTS

• **Human-centered computing → HCI → Interaction paradigms → Graphical user interfaces.**

KEYWORDS

Graphical user interfaces, notation, toolkits, software design

---

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

IHM '19 Adjunct, December 10–13, 2019, Grenoble, France.

© 2019 Copyright is held by the owner/author(s).

ACM ISBN 978-1-4503-7027-1/19/12.

https://doi.org/10.1145/3366551.3370352

RESUME

La programmation des interfaces graphiques reste un exercice laborieux et consommateur en temps l'essentiel de l'interaction devant généralement être codé via un langage de programmation. Ce code souvent verbeux et peu lisible impacte la réutilisabilité des interfaces et la conception itérative, la moindre évolution de l'interface nécessitant des modifications substantielles du code lors de sa conception. Nous proposons un modèle architectural et une boîte à outils expérimentale qui facilitent les modifications en les rendant peu coûteuses.

MOTS CLÉS

Interfaces graphiques, notation, boîtes à outils, conception logicielle

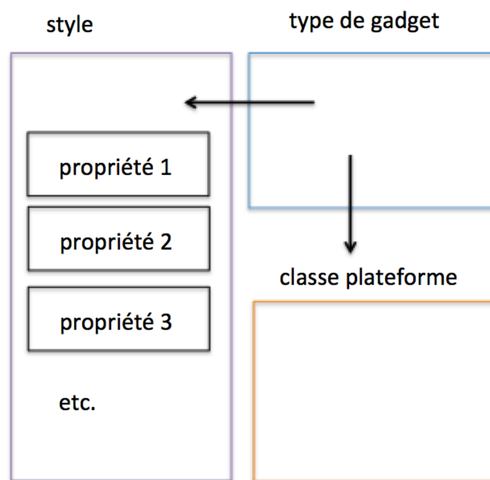
INTRODUCTION

En 1997, Myers et al. publiaient un article commençant ainsi "*Creating user interface software has proven to be very difficult and expensive because it is often large, complex, and challenging to implement, debug, and modify"* [5]. Vingt ans plus tard, la programmation des interfaces graphiques reste presque aussi complexe, l'essentiel de l'interaction devant généralement être codé au moyen d'un langage de programmation. Ce code, souvent verbeux et peu lisible, nécessite une connaissance étendue de l'interface de programmation (API) de la boîte à outils graphique (toolkit), et requiert un recours fréquent à la documentation, qui est souvent d'une taille considérable. Ainsi, malgré la popularisation des outils interactifs (e.g. générateurs d'interfaces) et des langages de spécification (typiquement en XML), la programmation d'interfaces nécessite toujours des compétences pointues et un temps de développement particulièrement important.

Un autre aspect, déjà souligné par Myers, est la difficulté à *modifier* les interfaces, d'où leur faible *réutilisabilité*. Ceci n'impacte pas seulement les évolutions futures, mais aussi le développement des systèmes interactifs. La conception centrée utilisateur requiert de multiples itérations, ce qui entraîne un coût considérable avec les toolkits actuels. En effet, la moindre modification de l'interface (par exemple changer un objet graphique en un autre afin d'occuper moins d'espace écran), peut entraîner de larges changements du code spécifiant l'interaction, voire nécessiter une réécriture intégrale. Elle peut aussi impliquer des complications imprévues, des objets graphiques apparemment équivalents ayant en fait des fonctionnalités différentes, ou manquantes.

Les outils interactifs (ou les descriptions XML) n'apportent pas de solution à ce problème dès lors que l'interface est fonctionnelle puisqu'ils spécifient essentiellement la structure et la présentation. Ceci conduit, soit à retarder l'implémentation de l'interaction (donc les tests utilisateurs avec un système fonctionnel), soit à augmenter significativement le coût de développement du fait des réécritures successives du code, soit, comme il est hélas facile de le constater, à fournir des interfaces sous-optimales pour tenir les contraintes de temps et de coût.

De notre point de vue, ces problèmes sont la conséquence d'une architecture logicielle inappropriée, et en particulier d'un usage excessif de la dérivation de classes, laquelle conduit à une explosion des



classes d'objets graphiques et à une divergence excessive de leurs fonctionnalités. Ceci a pour conséquence que des objets ayant des fonctionnalités de haut niveau similaires (par exemple du point de vue d'un designer) nécessitent un codage différent. Nous défendons ici l'idée que ce hiatus n'a pas de raison d'être, et que les objets fonctionnellement équivalents doivent se programmer de la même manière et être donc interchangeables. Ceci permet en outre de réduire le nombre de classes, donc la documentation et le temps consacré à la consulter.

Nous présentons ci-après une nouveau modèle architectural et un toolkit expérimental nommé *Guit*. Ce travail vise la simplicité d'écriture et la réutilisabilité du code. Cette communication se concentre sur la spécification de la structure et de la présentation, afin de faciliter les modifications et de les rendre peu coûteuses.

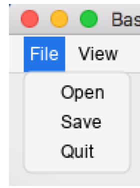
CLASSES PLATEFORMES

Guit repose sur une notion inspirée de l'industrie automobile, les *classes plateformes*. Les plateformes (ou châssis) permettent de concevoir et produire divers modèles de voitures à moindre coût car ils reposent sur la même infrastructure (une plateforme) et des composants qui sont en large partie interchangeables (moteurs, éléments de carrosserie, équipements internes, etc.). Ce modèle offre donc une grande flexibilité tout en rationalisant la production.

Guit procède de même. Un *gadget* (objet graphique) est la combinaison d'une *classe plateforme* (une classe au sens des langages de programmation) et d'un ensemble de composants appelés *propriétés*. Les classes plateformes servent principalement à stocker les propriétés et les enfants des gadgets, de manière à créer un graphe de scène. Les propriétés ne servent pas seulement à spécifier l'apparence graphique des gadgets, mais aussi les valeurs de leurs données (qui contrôlent ce qu'ils affichent), leurs états (visible, sélectionné, etc.), leurs comportements (édition de texte, sélection exclusive...), ainsi que les actions conditionnelles qui leurs sont associées (e.g. exécuter une fonction ou une transition d'état). Ce design permet une grande flexibilité tout en évitant une multiplication effrénée des classes dérivées, donc que les APIs divergent, car des gadgets fonctionnellement équivalents reposent sur les mêmes classes plateformes (ou des classes dont l'API est identique). Ils se programment donc de la même manière et sont donc interchangeables.

Pour des raisons de praticité évidente, les constructeurs proposent des *modèles* de voitures, c'est-à-dire des combinaisons prédéfinies de composants. Pour la même raison, *Guit* offre des *types de gadgets*. Un *type* est la combinaison d'une classe plateforme et d'un ensemble prédéfini de propriétés appelé *style*. Les instances de gadgets sont générées par les *types* et peuvent être personnalisées (par exemple pour afficher un texte donné ou changer un comportement) en leur ajoutant des propriétés ou des styles additionnels, ce qui a pour effet d'"effacer" les propriétés équivalentes de leur *type*. Ces styles additionnels peuvent être créés dynamiquement ou spécifiés par des feuilles de styles cascadées de type CSS, et être éventuellement partagés par plusieurs gadgets afin de faciliter la personnalisation des interfaces.

Les *types* et les *styles* peuvent être créés dynamiquement au cours de l'exécution. Cette caractéristique est intéressante car elle permet de générer des variantes d'une même interface à partir d'une même spécification, ce qui peut être vu comme une forme de plasticité [10].




```
auto& mymenu = Menu
<< On / [=](Gadget& g){doIt(g);}
<< "Open"
<< "Save"
<< "Quit";
```

Figure 1 : Spécification déclarative: Un menu contenant une expression conditionnelle et trois items. N'importe quel groupe d'objets s'écrit de la même manière.

```
auto& mymenubar = MenuBar
<< ("File" << mymenu)    // menu précédent
<< ("View"               // second menu
   << (Menu
       << ("Copy"
           << On / [=]{doCopy();}
          )
       << ("Delete"
           << On / [=]{doDelete();}
           << Color("red")
          )
      )
 );
```

Figure 2 : Spécification hiérarchique : Une barre de menu ouvrant le menu précédent et un second menu.

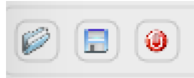
Une spécification paramétrée par des types déterminés à l'exécution peut ainsi par exemple générer des représentations plus ou moins compactes en fonction des contraintes de l'environnement (e.g. produire des menus d'items plutôt que des listes d'items).
Ce modèle architectural allie donc la configurabilité à la flexibilité. Il permet de plus d'accroître la réutilisabilité des interfaces, et, de ce fait, de faciliter le développement itératif. En effet, dans de nombreux cas, un ensemble de gadgets peut être trivialement remplacé par un autre sans avoir à modifier le code spécifiant l'interaction. Par exemple, toutes les constructions permettant d'effectuer un choix dans une liste (liste d'items, menus d'items, groupes de boutons, etc.) sont constituées d'items ayant un même parent. Comme expliqué plus bas, ces constructions se programment de la même manière, elles sont interchangeables et il suffit de changer le *type* du gadget parent pour obtenir la représentation souhaitée.
Ce modèle architectural sépare fortement la structure des propriétés, dans l'esprit de HTML/CSS, sauf que la notion de propriété est plus générale (comportements, actions conditionnelles, données des gadgets). Il repose sur la réification du rendu et du comportement via les propriétés, dans l'esprit de l'interaction instrumentale [1]. Il a aussi pour avantage de réduire l'empreinte mémoire car les gadgets n'ont pas de champs prédéfinis mais reposent sur les propriétés de leurs types (qui sont partagées) pour assurer leur comportement et leur rendu par défaut. Il exploite entre autres les patrons de conception *décorateur* et *injection de dépendance* [6] afin d'éviter un recours immodéré à la dérivation. Il présente aussi certaines similarités avec ECS [7], les propriétés pouvant être vues comme des composants ou des systèmes. La version actuelle du toolkit est implémentée en C++17, mais ce modèle n'est pas propre à un langage informatique donné.

SPECIFICATION DECLARATIVE ET ADAPTABILITE

Les spécifications *Guit* sont interprétables mais aussi *nativement* compilables en C++17. Cette second possibilité offre plusieurs avantages : 1) elle évite d'avoir à coder les interfaces en plusieurs lieux (typiquement, une spécification de la structure en XML et un codage de l'interaction avec un langage de programmation) et supprime de ce fait la nécessité d'avoir à écrire du "code glue" pour lier ces parties, 2) elle permet de détecter la plupart des erreurs précocement (i.e. à la compilation), et de réduire d'autant le débogage, 3) elle permet de bénéficier de toute la puissance du langage de programmation. Ceci permet par exemple de paramétrer la spécification (e.g. d'avoir un nombre d'objets arbitraire grâce à une boucle ou de configurer dynamiquement les types de gadgets), de la décomposer en classes, de traiter de la même manière les parties statique et dynamiques de l'interface (e.g. impliquant la création ou la destruction d'objet) ou encore d'associer ou changer facilement et sans risque d'erreurs des comportements ou des fonctions de callback.
Bien que les spécifications *Guit* soient compilables, leur forme est *déclarative* [2] afin de les rendre plus claires et plus compactes, grâce à la surcharge d'opérateurs infixes. Ces opérateurs sont *polymorphiques* : leur sens de haut niveau est toujours le même mais les opérations bas niveau qu'ils implémentent dépendent de leurs opérandes. Ces operateurs permettent une écriture plus "naturelle", car infixe, comme les formules mathématiques et ils remplacent quantité de fonctions, ce qui évite d'avoir à consulter la documentation pour trouver le nom de la fonction désirée.

```
auto& mybox = ButtonBox
<< On / [=](Gadget& g){doIt(g);}
<< "Open"
<< "Save"
<< "Quit";
```

Figure 3 : Gadget auto-adaptatifs : la spécification est la même qu'a la FIgure 1 à part le type du conteneur.



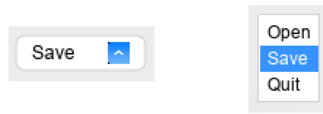
```
auto& mylist = ChoiceBox(".list")
<< On / [=](Gadget& g){doIt(g);}
<< "#open"
<< "#save"
<< "#quit";

/* feuille de style */
.list {dir: row; border: rounded;}
#open {icon: open.jpg;}
#save {icon: save.jpg;}
#quit {icon: quit.jpg;}
```

Figure 4 : Configuration des propriétés : la même spécification peut afficher du texte, des icones ou les deux suivant la feuille de style. Le texte peut être internationalisé de cette manière. .list et #open reférent à des styles, partagés (premier cas) ou individuels (second cas).

L'exemple de la Fig. 1 montre comment créer un groupe de gadgets, ici un menu contenant 3 items et une expression conditionnelle appelant une fonction de callback lorsqu'un item est cliqué. L'*opérateur <<* est l'opérateur d'ajout et il peut être chaîné, de telle sorte que **`a << b << c`** signifie que b et c sont ajoutés à a (dans cet ordre). Cet opérateur permet d'ajouter n'importe quel autre élément (propriété, callback, gadget si le parent est un conteneur) à un gadget. Il remplace à lui seul des dizaines de fonctions *addXXX()* ou *setYYY()* et il est déjà utilisé dans la bibliothèque standard C++ pour lire et écrire des données dans des flux d'entrée/sortie avec un sens et une syntaxe similaires. Une hiérarchie d'objets peut-être créée en combinant cet opérateur avec des parenthèses (Fig. 2).
Une caractéristique importante est que le *type* des items n'a pas besoin d'être spécifié. Les chaînes de caractères ("`Open...`") sont en effet automatiquement converties en *Items*, des gadgets qui ont la particularité d'être *auto-adaptatifs*. Un *Item* est un gadget qui déclenche une action lorsque l'on clique dessus (ou lorsqu'on le sélectionne/désélectionne s'il a un état). Une même et unique classe plateforme sert à produire tous les *types* d'items ou de boutons, la représentation ne dépendant que du style du *type* choisi. Le type *Item* a de plus pour particularité que son style s'adapte automatiquement au parent de l'item. Ainsi, suivant son parent un *Item* peut avoir l'apparence et le comportement d'un bouton avec ou sans état, d'un item de liste, de menu, de menu d'options, voire même d'un élément de "bar chart". Cette caractéristique a pour conséquence qu'un groupe de gadgets peut être modifié juste en changeant le nom de son conteneur. Par exemple le menu de la Fig. 1 peut être changé en boîte de boutons juste en changeant `Menu` en `ButtonBox` (Fig. 2). Il pourrait de même être changé en un groupe de boutons exclusifs (`RadioBox`), une liste d'items (`ListBox` ou `ChoiceBox`), un menu d'items (`OptionBox` ou `ComboBox`), etc. Ces boutons/items peuvent de plus afficher du texte et/ou une icone, être animés ou configurés différemment selon qu'ils sont ou non sélectionnés (e.g. afficher des textes différents avec des couleurs différentes).
Cette caractéristique contraste fortement avec les toolkits usuels, qui offrent profusion d'items ou boutons différents suivant qu'ils ont ou non un état, affichent du texte ou une icone, et surtout, suivant le conteneur auquel ils sont ajoutés. Ceci a pour conséquence que le moindre changement de l'interface peut nécessiter d'avoir à réécrire tout le code contrôlant l'interaction (ou la création de ces objets). Par exemple, changer l'équivalent d'un `ListBox` en `OptionBox` afin d'occuper moins d'espace écran, ou en `ChoiceBox` affichant des icones représenterait un travail non négligeable avec des toolkits tels que Cocoa, Qt ou Java Swing.

CONFIGURATION ET PARAMÉTRAGE DE LA SPÉCIFICATION

Les propriétés des gadgets peuvent être configurées à l'aide de feuilles de style, par exemple pour afficher des icones plutôt que du texte (Fig. 4), ce qui permet de produire plusieurs variantes de la même interface selon la feuille de style utilisée. Les données étant aussi des propriétés, le texte peut être internationalisé via les feuilles de styles. Les *types* étant des variables, ils peuvent être déterminé à l'exécution, ce qui signifie qu'une même spécification peut produire différents gadgets




```
BoxType& newType(bool is_mobile) {
  if (is_mobile)
    return OptionBox.clone(".mybox1");
  else
    return ListBox.clone(".mybox2");
}

auto& MyBox = newType(is_mobile)

auto& mybox = MyBox
<< On / [=](Gadget& g){doIt(g);}
<< "Open" << "Save" << "Quit";
```

Figure 5 : Paramétrage des gadgets : MyBox est un type different (OptionBox ou ListBox) suivant que le programme est ou non lancé en mode "is_mobile".

```
Strings items{"#open","#save","#quit"};
auto& mybox = MyBox
<< On / [=](Gadget& g){doIt(g);}
<< items;
```

Figure 6 : Paramétrage des items : Les items peuvent être spécifiés dans une liste de taille arbitraire.

```
auto& mybtn = Button("Change State")
<< On / (state1 >> state2)
<< On / (state2 >> state3)
<< On / (state3 >> state1);
```

Figure 7 : Transitions d'états : Cliquer sur le bouton permet de passer de l'état 1 à l'état 2, puis à l'état 2 à l'état 3, puis de nouveau à l'état 1, et ainsi de suite.

selon le contexte. Par exemple un gadget occupant moins d'espace est utilisé en mode *"is_mobile"* dans l'extrait de la Fig. 5. Un nouveau type (`MyBox`) utilisant un style dédié est créé dynamiquement à partir d'un type prédéfini. Ce type d'expression ne compile que s'l fait sens, ce qui assure l'intégrité de l'interface. Les items des conteneurs peuvent aussi être spécifiés à l'aide d'un tableau (Fig. 6), ce qui leur permet d'avoir un nombre arbitraire d'items déterminé à l'exécution.

EXPRESSIONS CONDITIONNELLES

Les exemples présentés dans ce papier ne présentent que le cas le plus basique : l'appel d'une fonction de callback. La syntaxe **`trigger / action`** signifie qu'une action est exécutée (ici une fonction de callback) lorsqu'un un "trigger" (ici un événement) est activé. Elle permet également de déclencher des transitions d'états (Fig. 7) sous une forme concise proche de [3].
`On` représente un événement par défaut qui dépend du *type* du gadget. Cet événement s'adapte si le type est adaptatif ou si le gadget est un conteneur. Ainsi, bien que la spécification soit la même, les fonctions seront appelées lorsqu'un item est cliqué dans le cas d'un menu, mais lorsque qu'il est sélectionné dans le cas d'une liste (ou dans tout autre conteneur exclusif). Cette propriété contribue également à la réutilisabilité des interfaces. Des événements spécifiques peuvent aussi être spécifiés (par exemple `On.mousedown`, `On.change`, `On.select`, etc.) si nécessaire.
Enfin, les triggers ne sont pas nécessairement des événements système et les propriétés sont des modèles (au sens de MVC) qui peuvent être partagées par plusieurs gadgets. Ceci permet de les synchroniser et de contrôler la logique de l'interface via les propriétés, en faisant quasiment abstraction des gadgets.

CONCLUSION ET PERSPECTIVES

Guit a été conçu de manière itérative, à partir de l'expérience de son concepteur et d'un certain nombre d'exemples types trouvés dans les manuels de programmation. Pour chaque nouvel exemple, le but a été de trouver une façon simple, générique, et facilement modifiable, d'obtenir le résultat souhaité. Sa forme actuelle n'est pas définitive, l'objectif étant de la faire évoluer en fonction des besoins et des retours des lecteurs et futurs utilisateurs. La facilité d'utilisation du toolkit n'a pas encore été testée, mais sa généricité, les simplifications qu'il permet et ses ressemblances avec HTML/CSS laissent à penser que sa prise en main devrait être plus facile qu'avec les toolkits habituels. Il permet aussi une réduction significative du nombre de lignes de code et du "fouillis syntaxique" (pas de "new", peu ou pas de variables intermédiaires et d'appels de fonctions nécessaires à la compilation mais sémantiquement inutiles).
Guit comporte d'autres aspects visant à simplifier l'interaction qui sont en cours de conception, en particulier l'utilisation de contraintes et la notion d'événement généralisé. Ces aspects feront l'objet d'une autre publication comprenant une analyse plus détaillée de l'état de l'art et une évaluation de l'utilisabilité du toolkit. L'objectif final est d'intégrer diverses idées, anciennes ou nouvelles, dans un tout cohérent. Une adaptation à d'autres langages (e.g. Python) ou aux interfaces Web constituerait une perspective intéressante.

RÉFÉRENCES

[1] M. Beaudouin-Lafon. 2000. Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces. (2000), 446–453. DOI: http://dx.doi.org/10.1145/332040.332473

[2] E. Lecolinet. 2003. A Molecular Architecture for Creating Advanced GUIs. In Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology (UIST '03). ACM, New York, NY, USA, 135–144. DOI:http://dx.doi. org/10.1145/964696.964711

[3] M. Magnaudet, S. Chatty, S. Conversy, S. Leriche, C. Picard, D. Prun. 2018. Djnn/Smala: A Conceptual Framework and a Language for Interaction-Oriented Programming. Proc. ACM Hum.-Comput. Interact. 2, EICS, Article 12, 27 pages. DOI: https://doi.org/10.1145/3229094

[4] B. A. Myers. 1991. Separating application code from toolkits: Eliminating the spaghetti of callbacks. In Proceedings of the ACM UIST. Addison-Wesley, 211–220.

[5] B. A. Myers, R G. McDaniel, R. C. Miller, A. S. Ferrency, A. Faulring, B. D. Kyle, A. Mickish, A. Klimovitski, P. Doane. 1997. The Amulet Environment: New Models for Effective User Interface Software Development. IEEE Trans. Softw. Eng. 23, 6, 347-365. DOI: https://doi.org/10.1109/32.601073.

[6] Patterns: https://en.wikipedia.org/wiki/Decorator_pattern et https://en.wikipedia.org/wiki/Dependency_injection

[7] T. Raffaillac, S. Huot. 2018. Application du modèle Entité-Composant-Système à la programmation d'interactions. 2018 . Conférence francophone sur l'Interaction Homme-Machine, Oct 2018, Brest, France. 42-51,

[8] T. Raffaillac, S. Huot. 2019. Polyphony: Programming Interfaces and Interactions with the Entity-Component-System Model. Proc. ACM Hum.-Comput. Interact. 3, EICS, Article 8, 22 pages. DOI: https://doi.org/10.1145/3331150

[9] S. Rey, S. Conversy, M.Magnaudet, M. Poirier, D. Prun, J-L. Vinot, S. Chatty. 2015. Using the Djnn Framework to Create and Validate Interactive Components Iteratively. In ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '15). ACM, New York, NY, USA, 230–233. DOI:http://dx.doi.org/10.1145/2774225. 2775438

[10] D. Thevenin, J. Coutaz, G. Calvary. 2003. A Reference Framework for the Development of Plastic User Interfaces. In Multi-Device and Multi-Context User Interfaces: Engineering and Applications Frameworks .