A

QUELQUES ÉLÉMENTS DU DÉVELOPPEMENT LOGICIEL

LA DÉCOUPE MVC (MODEL VIEW CONTROL)

Imaginez la programmation en Python d'un petit menu d'une application visible sur la figure A.1. Lorsqu'on clique sur un élément du menu, il ne se passe rien, ou presque : une fenêtre apparaît contenant juste un bouton « Do nothing button ». Totalement inutile, certes, mais c'est pour l'exemple.

Figure A.1 *Une application avec juste un menu*

Le code Python dont l'exécution affiche ce menu est repris ci-dessous, mais il est inutile à ce stade-ci d'en comprendre le fonctionnement en détail.

```
from Tkinter import *

def donothing():
  filewin = Toplevel(root)
  button = Button(filewin, text="Do nothing button")
button.pack()

root = Tk()
menubar = Menu(root)
filemenu = Menu(menubar, tearoff=0)
```

```
filemenu.add_command(label="New", command=donothing)
filemenu.add_command(label="Open", command=donothing)
filemenu.add_command(label="Save", command=donothing)
filemenu.add_command(label="Save as...",command=donothing)
filemenu.add_command(label="Close", command=donothing)

filemenu.add_separator()

filemenu.add_command(label="Exit",command=root.quit)
menubar.add_cascade(label="File",menu=filemenu)
editmenu = Menu(menubar, tearoff=0)
editmenu.add_command(label="Undo",command=donothing)

editmenu.add_separator()

editmenu.add_command(label="Cut",command=donothing)
editmenu.add_command(label="Copy",command=donothing)
editmenu.add_command(label="Paste",command=donothing)
editmenu.add_command(label="Delete",command=donothing)
editmenu.add_command(label="Select All",command=donothing)

menubar.add_cascade(label="Edit",menu=editmenu)
helpmenu = Menu(menubar, tearoff=0)
helpmenu.add_command(label="Help Index",command=donothing)
helpmenu.add_command(label="About...",command=donothing)
menubar.add_cascade(label="Help", menu=helpmenu)

root.config(menu=menubar)
root.mainloop()
```

L'affichage d'un tel menu ainsi que son utilisation exigent la prise en compte de trois fonctionnalités logicielles assez distinctes (mais pourtant en partie mélangées dans le code):

- Le graphisme du menu (par exemple sa couleur, le nombre d'éléments qui apparaissent suite à l'activation du menu, le type de caractère utilisé pour les éléments du menu).
- Les éléments qui composent le menu.
- Le fonctionnement du menu (qu'advient-il lorsqu'on choisit un des éléments).

On conçoit aisément que, même si le petit code affiché ne le fait délibérément pas, il est préférable de tenir séparés ces trois aspects. Ainsi, on pourrait décider de modifier la liste des éléments sans pour autant qu'il soit nécessaire de modifier et la représentation du menu et l'exécution associée à certains des éléments. Il en va de même pour cette représentation. On devrait pouvoir aisément changer la couleur du menu sans affecter en rien son fonctionnement. On pourrait même récupérer la liste des éléments pour un tout autre composant graphique, un ensemble de boîtes à cocher par exemple.

Il en découle naturellement qu'il est bien plus logique et bien plus aéré d'avoir un code se découpant comme ci-dessous : d'abord la liste contenant tous les élé-

ments, ensuite une boucle permettant de créer les éléments graphiques associés à ces éléments.

```
menuItems = ["New", "Open", "Save", "Save as...", "Close"]

root = Tk()
menubar = Menu(root)
filemenu = Menu(menubar, tearoff=0)

for x in menuItems:
    filemenu.add_command(label=x, command=donothing)
```

Déjà, pour cet exemple, et aussi modeste soit-il, la séparation de ces trois aspects et fonctionnalités logicielles n'en reste pas moins une première mise en œuvre de la recette de conception MVC. Ici le *Model* serait la liste des éléments (que l'on pourrait installer dans un fichier séparé), la *View* serait la représentation graphique de ce menu (type de composant graphique pour afficher la liste, disposition géométrique du menu, couleur et taille des éléments), et son *Control* le mode du fonctionnement du menu et des éléments qui lui sont associés (dans le code, la déclaration séparée des fonctions `donothing()`).

Un autre développeur pourrait vouloir récupérer votre formidable menu mais pour une toute autre liste d'éléments. Imaginez le cas d'un restaurateur chinois qui voudrait récupérer le menu d'un restaurateur italien. Même si, en général, il est impossible de commander le plat numéro 257 dans un restaurant italien, pourtant et depuis des années notre préféré dans les restaurants chinois.

USE CASE ET MVC QUI EN DÉCOULE

Depuis l'avènement d'UML (*Unified Modeling Language*) comme langage de modélisation graphique des développements logiciels, tout projet informatique débute généralement par la mise au point du diagramme *use case* (dit « cas d'utilisation » en français). Ce diagramme essentiel a pour mission d'identifier les différents usages de ce logiciel vu du côté utilisateur, il en compose son « cahier de charge » et il est au départ de tout. La figure qui suit illustre un tel diagramme dans le cas d'un distributeur automatique d'argent.

Décrivons la figure A.2. Ce logiciel sera utilisé par trois types d'acteur : « le client », « la banque » et « l'employé de maintenance ». Chacun de ces acteurs, et c'est ce qui les différencie, en fera un ensemble d'usages (appelés les « cas d'utilisation ») qui lui sont propres. Ainsi le client pourra « retirer de l'argent » ou « effectuer un virement » mais pas, et c'est tant mieux, effectuer la maintenance.

LE MODÈLE MVC ET USE CASE

Comme l'indique la figure A.3, ces use case vont donner naissance à trois types de fonctionnalité logicielle : contrôle, entité et interface, qui reprennent parfaitement la recette de conception MVC (il suffit d'effectuer la substitution Modèle = Entités

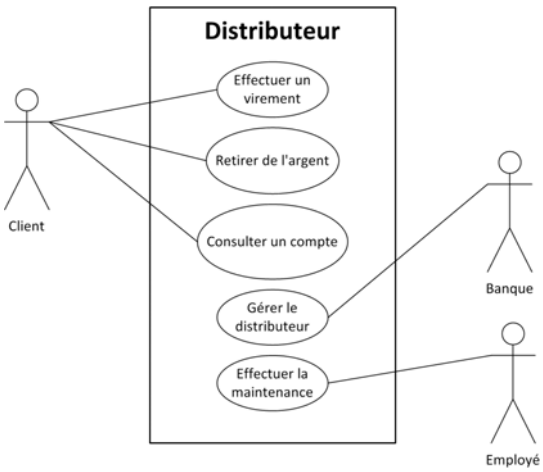

Figure A.2 *Diagramme « use case » d'un distributeur automatique d'argent*

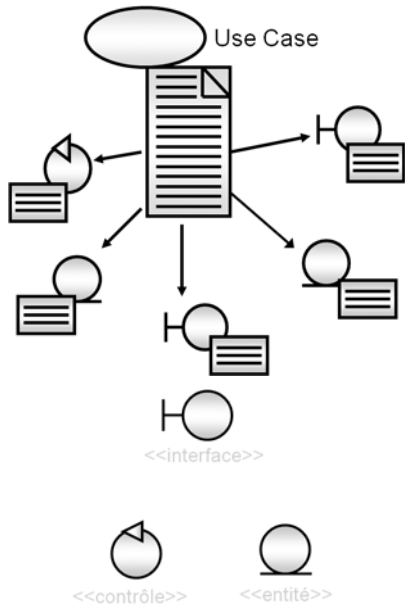

Figure A.3 *Le modèle MVC et use case*

= Modèle et Interface = Vue). Chaque cas d'utilisation sera pris en charge par les éléments contrôles. Ainsi, effectuer un virement sera la responsabilité d'un ensemble d'acteurs logiciels particuliers qui n'aura ni à se préoccuper de l'interface avec le client ni avec les bases de données et les éléments entités. Le contrôle se devrait de fonctionner de façon identique si l'on change la manière dont le client

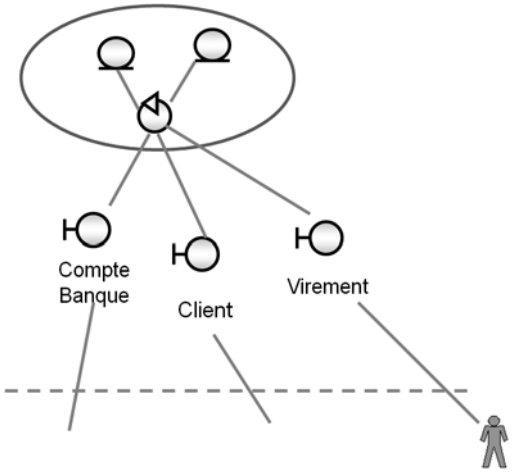

Figure A.4 *MVC et distributeur d'argent automatique*

voit sa banque sur son navigateur Web et la manière dont les comptes en banque sont finalement stockés sur le disque dur. À chaque intersection entre les acteurs et le cas d'utilisation doit correspondre un ensemble d'éléments interfaces (ou *view*) dont le rôle sera en effet de gérer l'interface graphique avec l'utilisateur.

Enfin, les objets fondamentaux dits parfois « métiers » utilisés par l'application tels que « CompteEnBanque », « Virement », « Client », etc. feront chacun l'objet d'une classe ou d'éléments logiciels qui, in fine, s'associeront à des versions permanentes de chacun d'entre eux (par exemple des tables « clients » et « compte en banque » dans une base de données relationnelles) stockés sur le disque dur. Comme la figure A.4 l'indique, les éléments contrôles demeurent au centre de tout. Ce sont les éléments pivots de l'application. Ils mènent la danse. Les interfaces, tout en recevant les sollicitations des utilisateurs, s'adressent à eux pour relayer celles-ci. À leur tour, les contrôles parlent aux éléments entités (par exemple, ils vont aller chercher le statut des comptes en banques ou créer de nouveaux virements) et sollicitent en retour les éléments interfaces pour recevoir de la part des utilisateurs d'autres informations utiles à la continuation de l'interaction (combien retirer du compte, sur quels autres comtes effectuer le virement, etc.).

Dans la plupart des environnements de développement logiciel, ces trois rôles, modèle/vue/contrôle, sont clairement tenus séparés et joués respectivement par les classes « Modèles », qui s'occuperont de la mise en correspondance avec la base de données relationnelles, les « classes GUI » (dans le cas d'un développement Web, il s'agira alors de mettre en œuvre ses connaissances d'HTML, CSS ou XML), qui permettront l'interface entre l'utilisateur et le système, et les fonctionnalités « contrôle » qui, par l'intermédiaire de codes (écrits dans n'importe quel langage de programmation), s'occuperont de toute la logique de fonctionnement de l'application.

Modélisation de processus et développement logiciel

À côté des diagrammes d'utilisation (use case) d'UML, l'industrie informatique a proposé un autre standard de notation destiné plus spécifiquement à modéliser les processus métier (ou *business process*), baptisé BPMN (*Business Process Modeling and Notation*). Des logiciels spécialisés permettent de tracer graphiquement les séquences d'activités composant les processus, en ce compris leurs entrées, actions, et décisions. Au départ, de tels modèles n'étaient destinés qu'à informer le développement futur du logiciel supportant le processus. Mais de plus en plus, la démarche de développement logiciel s'intègre verticalement, de l'analyse des processus au développement, test et déploiement du système. Dès lors, les logiciels de type BPMN s'interfacent désormais (quand ils n'y sont pas complètement intégrés) avec les outils de développement logiciel et les systèmes d'intégration d'application (*enterprise application integration* ou EAI) qui seront présentés au chapitre 9. L'idée est de rendre aussi transparente et automatique que possible la transposition d'un modèle (*use case* en UML ou processus métier en BPMN) en code informatique correspondant. L'architecture des systèmes d'information d'entreprise a largement évolué dans un sens qui permet cette intégration et vise à donner au maximum la haute main du développement logiciel à ceux qui en ont besoin, c'est-à-dire aux utilisateurs eux-mêmes.