



UNIVERSITÉ DE
SHERBROOKE

IFT 592

Projet d'informatique I
Amélioration du Finder 10.6

Rapport final

JONATHAN ACEITUNO

Matricule 09187925

Sommaire

1	Introduction	5
2	Spécification des fonctionnalités	6
1	Affichage de la taille des éléments sélectionnés	6
1	Mise en contexte	6
2	Fonctionnalité envisagée	7
2	Regroupement des emplacements en espaces de travail	8
1	Mise en contexte	8
2	Fonctionnalité envisagée	10
3	Tiroir de propriétés omniprésent	10
1	Mise en contexte	10
2	Fonctionnalité envisagée	10
3	Étude technique	14
1	État de l'art de la modification du Finder	14
1	Automator, AppleScript	14
2	Plugin d'importation Spotlight	15
3	Plugin de menu contextuel	16
4	Services	16
5	Interface d'accessibilité	17
6	Intrusion dans un exécutable Cocoa	17
2	Choix	19
4	Architecture de l'API Cocoa	20
1	Objective-C	20
2	Notions dérivées de OpenStep	21
1	Packages et bundles	21
2	Frameworks	22
3	Liste de propriétés	22
4	Constitution d'une application	22
3	Principaux frameworks	23
1	Foundation Kit	23
2	Application Kit	23

4	Processus type de développement rapide	24
1	Outils de développement	24
2	Définition des classes	24
3	Création de l'interface graphique	24
4	Mise en place des bindings	24
5	Écriture des contrôleurs	25
6	Le cas des délégués	25
5	Outils pour la <i>modification-singe</i>	27
1	Techniques de modification du programme à l'exécution	27
1	Catégories	27
2	Surcharge de méthode	28
3	Substitution de classe	28
4	Filoutage de méthode	29
5	Implémentation superséquente	30
6	Boxage de canard	30
7	Utilisation de l'écosystème de classes du programme cible	31
2	Exploration statique d'un exécutable Mac OS X	31
1	Récupération de données sur les classes	32
2	Lecture du code machine annoté	32
3	Extraction d'informations depuis les packages d'interface	33
3	Exploration dynamique d'un exécutable Mac OS X	33
1	Un langage pour communiquer avec Cocoa	33
2	Utilisation d'un débogueur	33
3	Utilisation de l'interface F-Script pour l'exploration	34
4	Création d'un bundle compatible SIMBL	34
1	Créer et placer le bundle	34
2	Faciliter le développement du bundle	35
5	Éthique	35
1	Conséquences du bris de certaines conditions	35
2	Attitude à adopter pour modifier	36
3	Survie dans le temps	36
6	Conclusion	36
6	Réalisation	38
1	Préférences d'application	39
1	Utilisation et mise en place des préférences	39
2	Interface graphique de choix des préférences	40
2	Pause pipi : gestion des fichiers dans le Finder	42
1	Famille FENode	42
2	Famille FINode	43
3	Famille NSNavNode	43
3	Affichage de la taille des éléments sélectionnés	44
1	Affichage d'un texte	45

2	Formatage du résultat et internationalisation	46
3	Récupération de la taille des éléments sélectionnés	46
4	Pause ca...fé : ajout de variables d'instance à des classes à l'exécution	47
5	Regroupement des emplacements en espaces de travail	48
1	Nature et comportement de la barre latérale	48
2	Enregistrement des informations de la barre latérale	48
3	Problèmes relatifs à la barre latérale	48
4	Lot de consolation	49
5	Réalisation	50
6	Tiroir de propriétés omniprésent	52
1	Fonctionnement des fenêtres et inspecteurs de propriétés	52
2	Conception de la vue du tiroir	53
3	Attachement du tiroir des propriétés aux fenêtres	54
4	Actualisation du contenu du tiroir	54
5	Récupération de l'icône d'un fichier	54
7	Packaging d'un plugin SIMBL	55
7	Conclusion	56
A	Bibliographie	58

Introduction

Le présent document a pour but de rassembler toutes les réflexions, justifications et recherches ayant fait l'objet des précédents rapports, qu'il inclut, ainsi que le détail des démarches qui ont été faites depuis le livrable préliminaire jusqu'à la date de fin du projet. Ce document se veut donc une référence unique pour tout le projet.

Nous commencerons par reprendre ce qui été établi dans la spécification des fonctionnalités **SPEC** afin de préciser le cahier des charges devant être respecté. Ensuite, nous reprendrons l'étude technique **TECH** qui détaille les moyens techniques pour arriver à ces fins, en étudiant chaque technologie susceptible d'être intéressante. Le livrable préliminaire **PREL** a ensuite servi de base pour compléter ce qui suit, à savoir une description de l'environnement de développement et de ses concepts, puis un tour d'horizon des techniques de modification que nous utiliserons, et enfin le détail de ce qui a été réalisé jusqu'au livrable final.

Les fichiers accompagnant le livrable sont disponibles à l'adresse suivante :

<http://oin.name/porcsharkfinder/>

Objectif du projet

Le système d'exploitation Mac OS 10.6 est fourni avec plusieurs applications de base, les *CoreServices*, qui sont intégrées à tout le système et exécutent des fonctions diverses. L'une d'entre elles, le *Finder*, est l'application de gestion de fichiers officielle. Cette application occupe cette place depuis le premier système Macintosh et a connu plusieurs bouleversements qui ont changé ses fonctionnalités à maintes reprises. Depuis la version 10.6 du système d'exploitation, le Finder a connu sa dernière grande modification puisqu'il a été réécrit en utilisant l'interface de programmation officielle Cocoa. Malheureusement, la réécriture n'a pas apporté beaucoup de nouveautés et certaines exigences des utilisateurs ne sont pas satisfaites depuis la création de Mac OS X. Le fait que le Finder est une application centrale des systèmes Macintosh en a fait un programme tentaculaire, intégré à diverses autres parties du système. Un remplacement total de ce programme est donc une solution sans issue. Cependant, la nouvelle version du Finder exploitant les technologies les plus modernes de Cocoa, il est possible de modifier son fonctionnement en lui rajoutant certaines fonctionnalités sans modifier le programme d'origine. L'objectif du projet est donc d'implanter trois nouvelles fonctionnalités facilitant un usage fréquent du Finder.

Spécification des fonctionnalités

Ce chapitre a pour but de décrire les souhaits pour les fonctionnalités du projet d'amélioration du Finder 10.6. Le lien entre les souhaits et la réalisation effective sera fait en temps voulu dans un chapitre ultérieur. L'utilité de chacune de ces fonctionnalités sera mise en lumière par un descriptif de mise en contexte.

Une fonctionnalité sera décrite par son utilité, sa place et sa représentation graphique dans le programme existant, et par les actions utilisateur qu'il est possible de réaliser en lien avec cette fonctionnalité.

SECTION



Affichage de la taille des éléments sélectionnés

1 Mise en contexte

Une des tâches courantes de la gestion de fichiers est la sélection et le tri dans un groupe de fichiers, que ce soit pour les supprimer ou pour les répartir sur différents espaces de stockage. En effet, malgré l'augmentation de la capacité des supports numériques de stockage et la disponibilité de formats de fichier à la compression toujours plus efficace, il est fréquent de s'approcher de la capacité maximum du support de stockage principal, et de devoir recourir à une sélection parmi les fichiers existants, ou à l'utilisation de supports de stockage auxiliaires comme les disques durs externes ou les clés USB. Ces supports secondaires sont également limités par une capacité de stockage et le gestionnaire de fichiers doit être un outil permettant de pouvoir rapidement obtenir de l'information sur les fichiers en jeu afin d'optimiser le temps de prise de décision, et par conséquent de réduire le temps total de cette tâche.

Le Finder possède deux modes de fonctionnement par fenêtre nommés *navigation*¹ et *spatial*², et l'utilisateur peut passer de l'un à l'autre en activant le bouton de barre d'outils

¹ C'est un mode hérité des gestionnaires de fichiers UNIX et Windows où une fenêtre représente une session de navigation et est agrémentée le plus souvent d'une barre latérale et d'une barre d'outils permettant la manipulation indirecte. Voir figure 1.1.

² Ce mode était le mode par défaut jusqu'à Mac OS X. Une fenêtre représente un unique répertoire et les

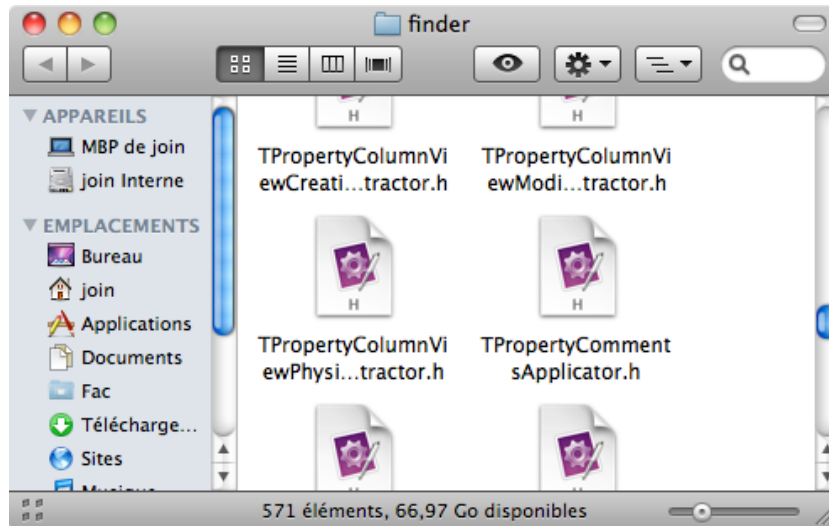


Figure 1.1. Éléments graphiques du mode navigation

en haut à droite d'une fenêtre Finder. À tout moment, dans chacun de ces deux modes, il est possible de lire dans la *barre de statut* des informations sur la sélection en cours et les capacités du support logique. De nombreux utilisateurs regrettent de ne pas pouvoir y lire la taille des éléments sélectionnés alors qu'il est possible d'en connaître le nombre.

Pour réaliser cette opération, un utilisateur doit ouvrir l'*inspecteur d'informations* par le raccourci `⌘⇧I`. Cette méthode présente de nombreux inconvénients puisque l'inspecteur d'informations n'est pas lié à une fenêtre : il est ainsi très difficile de pouvoir comparer visuellement les tailles respectives de deux sélections effectuées dans deux fenêtres différentes. De plus, l'opération est tellement courante qu'il est coutume, dans les autres gestionnaires de fichiers, que la taille des éléments sélectionnés soit visible à tout moment.

2 Fonctionnalité envisagée

Le projet devait disposer de la fonctionnalité suivante. Lorsqu'au moins un élément est sélectionné dans la fenêtre, alors le texte de la barre de statut³ doit inclure la taille totale de ces éléments dans une unité lisible. Le calcul de la taille totale de répertoires pouvant être parfois assez long, une option de type case à cocher pour activer ou non le calcul de la taille des sélections contenant des répertoires est disponible dans les options avancées des préférences de l'application Finder.

caractéristiques de la fenêtre (position, taille, présentation du contenu) sont différentes d'un répertoire à l'autre. Ce mode permet une manipulation directe des fichiers mais peut être handicapante lorsque la hiérarchie est complexe. Voir figure 1.2.

³ Ce texte ressemble habituellement à ``1 sur 7 sélectionné, 66,9 Go disponibles''.

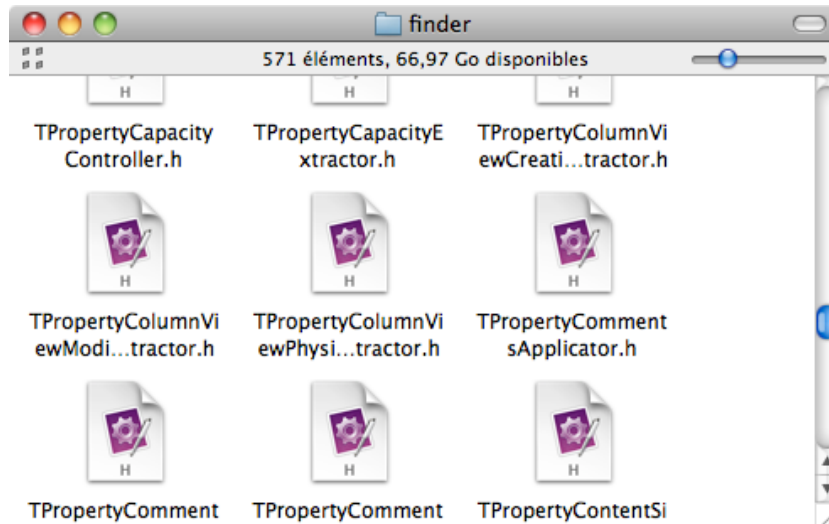


Figure 1.2. Éléments graphiques du mode spatial

SECTION

2

Regroupement des emplacements en espaces de travail

1 Mise en contexte

Le Finder permet uniquement de naviguer dans un système de fichiers hiérarchisé⁴, à l'instar de la grande majorité des gestionnaires de fichiers. L'inconvénient majeur de ce type de navigation est qu'il n'est pas aisé de retrouver des répertoires utilisés fréquemment en navigant uniquement dans la hiérarchie. De plus, il pourrait être utile d'avoir une organisation alternative mettant en jeu les répertoires les plus utilisés.

La solution proposée par le Finder depuis la version 10.3, est de proposer une barre latérale permettant d'y inclure des raccourcis vers les emplacements les plus courants. Elle est séparée en deux parties : les appareils en premier, puis ensuite les emplacements favoris. À partir de Mac OS 10.5, la barre latérale a évolué pour permettre d'utiliser quatre catégories : les appareils, les appareils sur le réseau, les emplacements favoris et les recherches. La présentation de ces catégories peut être configurée sommairement (afficher, cacher, développer ou réduire chaque catégorie) mais il n'est pas possible de les réorganiser, de les renommer ou d'en ajouter de nouvelles, ce qui aurait pu permettre d'organiser les emplacements favoris par espaces de travail et d'y avoir accès à n'importe quel moment.

⁴ Exception est faite des *dossiers intelligents* qui sont des moyens de naviguer dans des résultats de recherche : ces dossiers intelligents ne sont pas hiérarchisés et la navigation se fait en spécifiant des filtres pour réduire le nombre de résultats.



Figure 2.1. Barre latérale non hiérarchisée

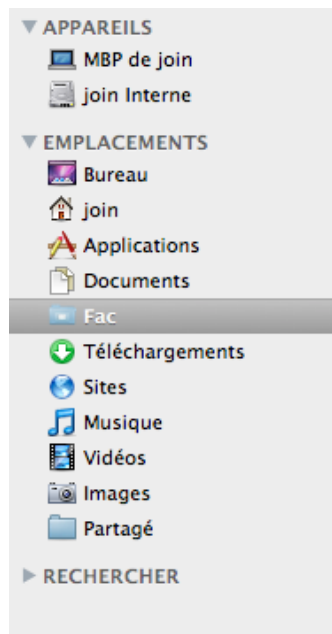


Figure 2.2. Barre latérale à un niveau hiérarchique

2 Fonctionnalité envisagée

Le projet devait disposer de la fonctionnalité suivante. La barre latérale doit pouvoir être configurée pour pouvoir :

- déplacer les catégories existantes, par glisser-déposer ;
- ajouter des catégories ;
- supprimer des catégories qui ne font pas partie des quatre catégories de base ;
- renommer des catégories.

Le remplissage de ces catégories par des éléments est une fonctionnalité implicite car elle est déjà présente.

En revanche, il faut savoir comment proposer les opérations d'ajout, de modification et de suppression des catégories, et cela dépendra de plusieurs paramètres et la décision ne sera réellement prise que lors de l'étude préalable à la réalisation de cette fonctionnalité.

Il est toutefois possible de donner quelques pistes :

- présence d'un bouton d'ajout de catégorie dans la barre de statut (barre horizontale tout en bas d'une fenêtre), à l'image de l'application iCal ;
- présence d'entrées dans un menu contextuel pouvant être appelé par un **Clic secondaire** ou un **^Clic** ;
- ou manipulation directe par l'entrée Barre latérale des préférences du Finder.

SECTION

3

Tiroir de propriétés omniprésent

1 Mise en contexte

L'utilisateur avancé d'un gestionnaire de fichiers doit quelquefois avoir accès à des propriétés sur les fichiers ou les répertoires qu'il explore. La vue en *colonnes* est la réponse du Finder à ces besoins. De plus elle permet de naviguer facilement dans la hiérarchie, tout en permettant de rapidement examiner les propriétés de l'élément sélectionné grâce à une dernière colonne les récapitulant. Chaque vue disponible a ses avantages et ses inconvénients pour l'organisation et le repérage rapide de fichiers, et les utilisateurs ont leurs propres préférences pour ces vues. Or, l'affichage instantané des propriétés n'est disponible qu'avec la vue en colonnes, à moins d'utiliser l'inspecteur de propriétés qui possède des inconvénients. Plusieurs autres gestionnaires de fichiers possèdent cet affichage pour chaque fenêtre de manière indépendante de la vue utilisée.

2 Fonctionnalité envisagée

Le projet devra disposer de la fonctionnalité suivante.

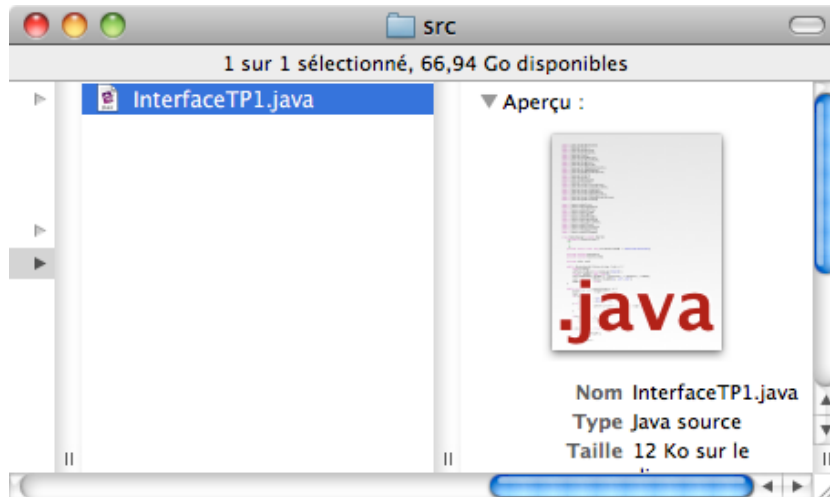


Figure 3.1. Colonne des propriétés

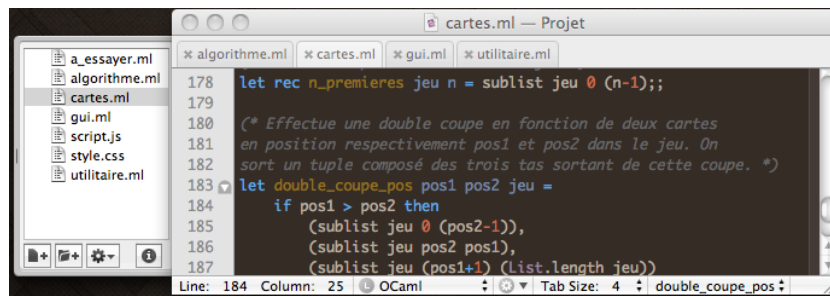


Figure 3.2. Exemple de tiroir dans l'application Textmate

a Définitions

Un **tiroir** est un élément graphique attaché à une fenêtre par la gauche ou par la droite, de même hauteur que la fenêtre d'attache, de longueur réglable, et fait pour contenir d'autres éléments graphiques. Il est possible de le cacher ou de l'afficher.



Il est conseillé d'utiliser l'élément tiroir dans le cas où il renfermerait des éléments d'interface fréquemment utilisés mais qui n'ont pas besoin d'être visibles la plupart du temps **HIG**.

b Description

Un tiroir doit être attaché à chaque fenêtre. Dans un tiroir, un panneau standard de propriétés sera affiché ⁵. Il affichera les propriétés de l'élément ou des éléments sélectionnés dans la fenêtre à laquelle est attaché son tiroir.

Une entrée dans le menu **Présentation** permettra d'afficher ou de cacher ce tiroir pour la fenêtre active. Il sera caché par défaut pour toutes les fenêtres.

Le positionnement du tiroir par rapport à sa fenêtre (à gauche, à droite) doit être l'objet d'un choix dans les préférences générales de l'application Finder. Si la fenêtre est positionnée vers un bord de l'écran de manière à cacher le tiroir, alors ce dernier doit être automatiquement positionné de l'autre côté.

⁵ En effet, l'affichage de propriétés sur un élément par une fenêtre de propriétés  ou un inspecteur de propriétés  fait appel au même panneau standard de propriétés.

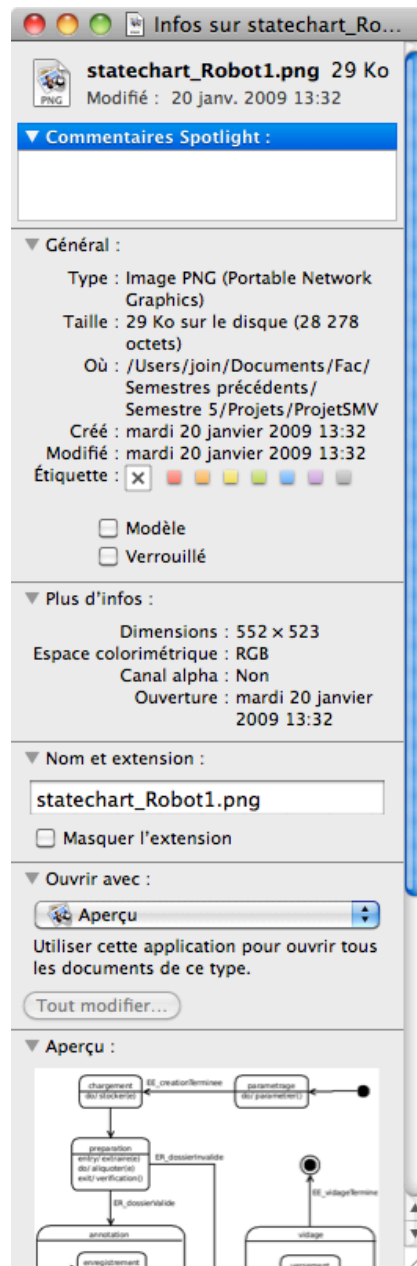


Figure 3.3. Fenêtre de propriétés

Étude technique

Ce chapitre a pour but de présenter un tour d'horizon des possibilités techniques de modification et un choix argumenté de certaines d'entre elles dans le cadre du projet d'amélioration du Finder 10.6.

Les choix pour les trois fonctionnalités présentées dans le rapport précédent **SPEC** sont motivés par une étude approfondie des possibilités de modification de l'application Finder à travers un état de l'art et un historique sur ce sujet.

SECTION



État de l'art de la modification du Finder

L'absence d'une interface de programmation officielle **SOFL** pour l'extension du Finder est compensée par une grande variété d'approches différentes. On peut expliquer cette variété en considérant le fait que les différents éléments de Mac OS X profitent d'une grande intégration.

1 Automator, AppleScript

a Description

Le langage de script officiel d'Apple, **AppleScript**, et l'application de construction graphique de processus automatiques, **Automator**, ont été créés pour automatiser des tâches en relation avec les interfaces des programmes. En effet, la description des actions est le plus souvent basée sur des éléments d'interface graphique des programmes. En associant ces actions à des éléments des langages de programmation comme les structures de contrôle ou les variables, il est donc possible de faire tout ce qui peut se faire manuellement à l'aide des périphériques d'entrée (clavier, souris) et de l'adapter à un contexte particulier. De plus, AppleScript permet l'écriture de boîtes de dialogue graphiques simples.

b Critique

Le langage AppleScript est assez proche d'une langue naturelle où les instructions sont énoncées à l'impératif. Quant à Automator, il s'agit d'un programme graphique favorisant la manipulation directe des instructions. Il est donc évident que ces solutions ont l'avantage d'être extrêmement simples à réaliser et de permettre tout ce qui peut être fait manuellement. La création d'une application à partir d'un Applescript est rendue triviale par l'éditeur officiel.

Néanmoins, il est impossible de faire plus que la simple utilisation de l'interface graphique des programmes compatibles (le Finder en fait partie) et la création de simples boîtes de dialogue. De plus, étant donné que son utilisation repose essentiellement sur le fait qu'on simule les entrées de l'utilisateur, il est parfois nécessaire de prévoir des temps d'attente, ce qui peut en faire une solution relativement lente, et si l'interface subit des modifications dans une version ultérieure, la compatibilité n'est pas garantie.

c Cas d'utilisation pertinent

Lors du travail d'un groupe de documents comme un projet de programmation, il est parfois utile de marquer les fichiers sur lesquels on a fini de travailler, ou ceux qui fonctionnent, afin de les différencier facilement de ceux qui ne fonctionnent pas **LABL**. Une solution simple est alors d'assigner une étiquette de couleur aux fichiers à marquer, de façon à les reconnaître au premier coup d'œil. Pour éviter une manipulation longue et hasardeuse (clic secondaire sur les fichiers à marquer, puis clic sur l'entrée de menu contextuel permettant d'assigner une étiquette de couleur rouge), il suffirait de créer un Applescript qui assigne cette étiquette automatiquement. Le script pourra ensuite être appelé facilement soit par un raccourci au clavier, soit par un clic ou un glisser-déposer s'il est rangé à un endroit accessible – barre d'outils du Finder, Dock ou barre latérale.

2 Plugin d'importation Spotlight**a Description**

Avec Mac OS 10.4, Apple a introduit un système d'indexation de fichiers nommé **Spotlight**, ayant pour but de recenser les données contenues dans un ordinateur (fichiers vus comme des images, documents, musiques, vidéos, applications, mais aussi messages électroniques, pages web récemment visitées...) avec leur méta-données, dans l'objectif de faciliter et d'accélérer les recherches. Une interface de programmation particulière est prévue depuis sa création, permettant ainsi de rajouter des modules enfichables permettant l'indexation de nouveaux types de contenu et leur intégration au reste de la base de données Spotlight. L'intégration au Finder se fait naturellement dans les zones où sont affichées des informations (panneaux et inspecteur d'informations) sur les fichiers, ainsi que dans l'utilisation des dossiers intelligents et de la fonctionnalité de recherche.

b Critique

La discussion des avantages et des inconvénients est limitée du fait que cette solution s'applique à une famille très particulière de problèmes et qu'elle n'est pas adaptée lorsqu'il n'est pas question de l'intégration d'un nouveau format de fichier au système.

c Cas d'utilisation pertinent

Lorsqu'une nouvelle application voit le jour, avec un nouveau format de fichier, il est commode de réaliser un plugin d'importation Spotlight afin de favoriser l'intégration de l'application avec le Finder.

3 Plugin de menu contextuel**a Description**

Une des maigres possibilités qu'Apple a introduites pour modifier directement un point de l'interface du Finder est l'interface de programmation des plugins de menu contextuel. C'est le menu que l'on obtient lorsqu'on effectue un clic secondaire sur des sélections dans le Finder. Un plugin de menu contextuel rend possible l'ajout dynamique d'entrées dans ce menu. Les entrées de menu peuvent être fonction des fichiers sélectionnés et la seule action possible est de cliquer dessus pour exécuter une action.

b État actuel

La réécriture du Finder depuis Mac OS 10.6 a entraîné la disparition des plugins de menu contextuel **CMPX**. Ils sont dorénavant remplacés par les services, mieux intégrés qu'auparavant et disponibles pour toutes les applications.

4 Services**a Description**

Les services sont présents dans toutes les applications des systèmes dérivés de NEXTSTEP depuis sa création. Mac OS X en fait partie depuis sa première version. Le principe d'un service est d'offrir une fonctionnalité contextuelle (par exemple, redimensionner une image) pour un objet (par exemple, une image sélectionnée dans un éditeur d'image). L'intérêt est que les applications qui offrent les services n'ont pas besoin d'être démarrées pour que le service puisse être proposé. Les services vont dans le sens d'une utilisation de l'ordinateur orientée vers le contenu. En contrepartie, les services n'ont jamais été mis en valeur.

Une légère amélioration a été faite dans Mac OS 10.6 puisque les services jouent maintenant le rôle supplémentaire des plugins de menu contextuel. Dans n'importe quelle

application, Finder y compris, selon le contexte voulu, des services sont disponibles lors de l'appel du menu contextuel.

b Critique

Les services ne sont pas faits pour être dynamiques et ne peuvent pas réellement concurrencer les plug-ins de menu contextuel du Finder, dans leurs nouvelles fonctions **SPRY**.

5 Interface d'accessibilité

a Description

Depuis la version 10.2, Mac OS X fournit une interface de programmation pour l'accessibilité **ACCS**. Les applications les plus courantes l'utilisent et fournissent ainsi une manière différente de communiquer avec les éléments de l'interface graphique. Le Finder fait partie des applications qui peuvent être utilisées à travers l'interface d'accessibilité.

Un programme fourni avec les outils de développement officiels ¹ permet d'explorer ce qui peut être vu et modifié en utilisant l'accessibilité.

b Critique

L'interface de programmation existante ne permet guère plus que de consulter les éléments de l'interface de n'importe quelle application de manière hiérarchique et d'agir sur ces éléments d'interface en simulant des actions des périphériques d'entrée ou en entrant directement des valeurs. Cette approche, bien que plus viable, subit les mêmes critiques que l'utilisation des Applescripts.

6 Intrusion dans un exécutable Cocoa

Les solutions présentées dans cette section ont toutes en commun la méthode : il s'agit d'exécuter le module voulu au lancement d'une application afin de pouvoir y rajouter du code. Il existe plusieurs techniques pour rajouter du code de manière fiable et durable connaissant le diagramme de classes d'une application : le *class posing* (faire passer une classe de sa conception par une classe déjà connue du programme) et le *method swizzling* (changer la table des sélecteurs Objective-C pour faire passer une méthode de sa conception pour une méthode déjà connue, en choisissant de pouvoir appeler ou non l'ancienne méthode au moment opportun). Le principe est qu'avec des modifications mineures et dans le respect de certaines règles de conception, il est possible d'intervenir sur des classes préexistantes sans causer de problèmes. Bien entendu, l'intrusion dans un

¹ Dans une installation normale, il se situe ici : `/Developer/Applications/Accessibility Tools/Accessibility Inspector`.

exécutable Cocoa nécessite de connaître beaucoup de détails de sa conception par rétro-ingénierie². De plus, une restructuration du code des programmes entraîne l'inefficacité des modules écrits pour l'ancienne structure. C'est ainsi qu'on peut qualifier cette approche d'interne, en comparaison avec des interfaces de programmation externes et des langages de script. Cependant, l'histoire des plugins de menu contextuel a montré que ces approches n'étaient pas non plus à l'abri d'une restructuration.

a InputManager

Un *InputManager* est un programme enfichable destiné à proposer une manière alternative de gérer les entrées du clavier (par exemple pour le braille ou les caractères asiatiques) et de les afficher, et ce dans toute application **INPM**. Cette technologie est disponible depuis la première version de Mac OS X et a très souvent été détournée. En effet, le fait qu'un *InputManager* soit lancé comme partie de chaque application Cocoa permet de l'utiliser pour s'infiltrer dans la hiérarchie de classes d'un programme et y rajouter du code, permettant ainsi de rajouter des fonctionnalités dans une application existante.

Avant Mac OS 10.6, le Finder était écrit avec Carbon, ce qui empêchait l'utilisation des *InputManagers* pour l'utiliser. Depuis Mac OS 10.6, les *InputManager* sont considérés obsolètes par Apple et il n'est plus possible de les utiliser.

b Application Enhancer

Application Enhancer est une solution commerciale qui propose une interface de programmation complète destinée à la fabrication de *haxies*, c'est-à-dire de programmes modifiant le comportement des applications Cocoa. Cela revient à la solution précédente, exception faite du support technique et des facilités de gestion offerts par le constructeur.

Le kit de développement coûte 100 USD et, malgré l'existence d'une version allégée, la solution n'est disponible pour Mac OS 10.6 qu'en version de test.

c PlugSUIT

PlugSUIT est une application permettant de réaliser la même chose qu'*Application Enhancer* et propose les mêmes avantages du côté de l'utilisateur (présence d'un panneau de préférences permettant de gérer les modules installés).

La version actuelle est incompatible avec Mac OS 10.6.

d SIMBL

SIMBL est le nom d'une technologie open source similaire à *Application Enhancer*, et bien qu'elle soit moins supportée, elle a l'avantage d'être gratuite. Avant l'arrivée de Mac OS 10.6, SIMBL était l'acronyme de Smart InputManager Bundle Loader. L'obsolescence

² Le fait que les applications Cocoa soient réalisées dans le langage Objective-C arrange bien les choses, car c'est un langage très dynamique, ce qui impose que le programme exécutable contienne une quantité importante d'informations sur les classes du programme.

des InputManagers ayant forcé le développeur à changer de point d'entrée, l'acronyme a également changé de sens.

La version actuelle est totalement compatible avec Mac OS 10.6 et c'est actuellement l'unique et le meilleur moyen pour réaliser l'intrusion de code dans un exécutable Cocoa comme le Finder.



Choix

Il apparaît à la lumière de cette étude que parmi toutes les possibilités, seule l'intrusion de code permet de réaliser les fonctionnalités envisagées pour ce projet. En effet, ne disposant pas d'une interface de programmation et d'extension complète, il est impossible d'opérer en utilisant des méthodes externes pour rajouter des éléments d'interface graphique opérationnels et intégrés à l'existant. C'est pourtant ce que demande chacune des trois fonctionnalités choisies, à travers :

- l'intégration d'un tiroir aux fenêtres existantes, et la réutilisation d'une vue existante pour remplir ce tiroir ;
- la modification non prévue de la barre latérale ;
- la modification non prévue de la barre de statut.

La documentation disponible et les tests réalisés ont indiqué que SIMBL est un outil de choix pour mener à terme la réalisation du projet.

Architecture de l'API Cocoa

Ce chapitre a pour but de passer rapidement sur les principales choses à savoir sur le développement moderne d'applications sur Mac OS X. Il n'y a pas si longtemps, deux API coexistaient, l'une nommée Carbon, étant utilisée avec le langage C et compatible avec les dernières versions *pre-UNIX* de Mac OS ^❶, et l'autre nommée Cocoa, toutes les deux apparues avec Mac OS X. La dernière a enterré l'autre mais son origine remonte toutefois à une vingtaine d'années. En effet, Cocoa trouve ses origines dans le système d'exploitation NeXTSTEP et son API utilisant le langage Objective-C. L'évolution fit naître la spécification OpenStep, dont Cocoa était, au départ, l'API de l'implémentation tirée de celle de l'entreprise NeXT.

SECTION



Objective-C

Le langage de base pour écrire les programmes Mac OS X avec Cocoa est Objective-C. C'est un surensemble très léger du C qui lui rajoute le paradigme orienté objets dans un style inspiré de Smalltalk ^❷. Ainsi ce langage se démarque de C++ sur plusieurs points ^❸ et n'est clairement pas destiné à être aussi performant, mais il permet en contrepartie une programmation agréable et des fonctionnalités de haut niveau.

Il y a quelques particularités du langage dont il est bon d'être informé. Les classes sont définies dans un bloc `@interface` et les méthodes sont déclarées dans un bloc `@implementation` correspondant. Les classes peuvent être hiérarchisées selon un héritage simple, il existe des méthodes de classe, et les protocoles `@protocol` définissent des interfaces qui peuvent être implémentées par des classes selon un héritage multiple. Un

^❶ Il s'agit d'un surensemble réorganisé de l'API Toolbox précédemment utilisée sur les environnements Macintosh.

^❷ Voir <http://developer.apple.com/Mac/library/documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html>

^❸ C++ redéfinit quelques règles du C (et en rajoute beaucoup d'autres) quand Objective-C rajoute des règles (moins d'une vingtaine de nouveaux mots-clé). C++ est plutôt statique (polymorphisme explicite, templates...) alors que Objective-C est dynamique (typage et chargement dynamiques).

protocole peut être informel ou formel, le dernier cas signifiant qu'une classe l'implémentant doit obligatoirement le faire en totalité.

En effet, le système d'objets d'Objective-C est particulier et inspiré de Smalltalk, puisqu'il fait intervenir non pas des appels directs de méthodes, mais des envois de messages avec la syntaxe suivante `[objet methode]`⁴. Il est possible d'envoyer des messages au pointeur de l'objet en cours `self` dans une méthode, au pointeur d'objet nul `nil`, et un message envoyé à un objet ne prenant pas en charge le sélecteur (descripteur de méthode) demandé est ignoré⁵.

Le typage canard (*duck typing*⁶) est également possible puisqu'on peut référer à des objets non seulement par des pointeurs typés, mais également par le pointeur d'objet générique `id`. Cela permet d'interpréter le type d'un objet par sa sémantique, l'ensemble des méthodes qu'il implémente, plutôt que par héritage.

Enfin, Objective-C propose le système des catégories, qui permet de rajouter des méthodes à l'interface d'une classe préexistante.

SECTION
2

Notions dérivées de OpenStep

La tradition OpenStep a laissé plusieurs concepts clés qui sont encore utilisés aujourd'hui et présents dans tout le système d'exploitation.

1 Packages et bundles

Il est souvent commun de confondre les termes **package** et **bundle**, qui réfèrent pourtant à deux notions différentes **SIXC** qui ont pour point commun la perception que le système a d'eux. Dans les deux cas, on parle d'un répertoire du système d'exploitation, le plus souvent muni d'une extension.

Un **package** décrit donc la notion de répertoire vu comme une entité atomique. C'est de cette façon que plusieurs types de documents sont stockés⁷.

Un **bundle** est simplement un répertoire muni d'une structure particulière reconnue par le système, et tous les bundles d'un même type ont la même structure. Il est très courant qu'un bundle contienne un fichier exécutable.

⁴ Les descripteurs des méthodes, nommés *sélecteurs*, comprennent également les paramètres éventuels qui peuvent être nommés. Voici un exemple d'appel à une méthode comportant des paramètres nommés : `[chien aboyerSur :maurice deFaçon :mechante]`. Ici le prototype de la méthode s'écrit `-(void)aboyerSur :(id)laPersonne deFaçon :(id)laFaçon`.

⁵ Il se peut que le message soit réexpédié à un autre objet, mais cela concerne une pratique connue sous le nom de *forwarding* que je n'utiliserai pas explicitement.

⁶ L'idée du typage canard est résumée par le test du canard, dont l'expression aurait pour origine le poète James Whitcomb Riley : ``When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck''.

⁷ Citons à titre d'exemples les packages `.rtfd` (RTF avec pièces jointes) et les packages `.download` symbolisant un fichier en cours de téléchargement avec le navigateur Safari.

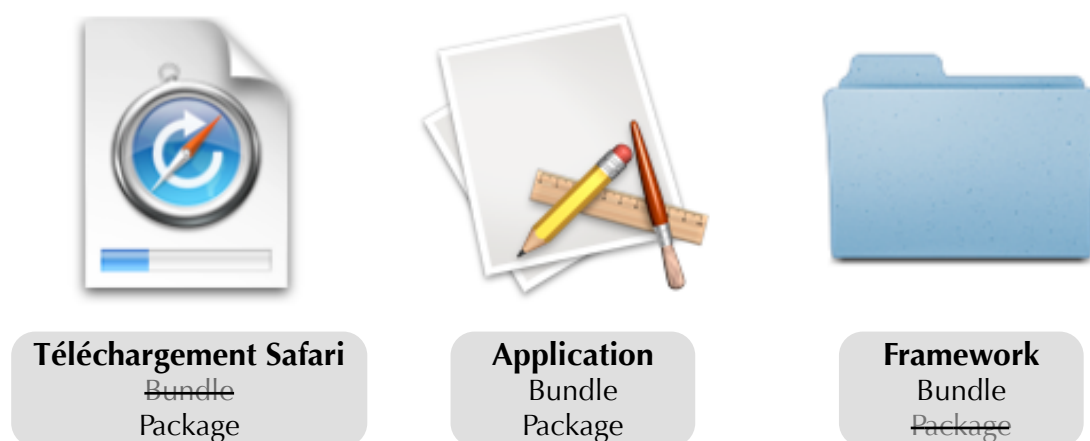


Figure 2.1. Exemples soulignant les différences entre package et bundle

Il peut exister des packages qui ne sont pas des bundles, comme certains types de documents, des répertoires qui sont à la fois package et bundle, comme les applications, et des bundles qui ne sont pas des packages, comme les frameworks.

2 Frameworks

Un **framework** est un bundle regroupant toutes les ressources nécessaires à l'utilisation d'une librairie dynamique. La structure d'un framework comprend les librairies, les fichiers d'en-tête et les autres ressources pour chaque version, ainsi qu'un lien vers la version par défaut. Cela simplifie le développement, la distribution et l'utilisation des librairies.

3 Liste de propriétés

Un format de fichier très courant est celui des **listes de propriétés** dont l'extension est `.plist`. Il existe plusieurs formats de listes de propriétés^⑧ mais le seul autorisé par Mac OS X est exprimé par XML et une DTD. Il est possible de se trouver confronté à des listes de propriétés encodées en binaire, pour des raisons diverses. Une liste de propriétés est très souvent définie comme un dictionnaire (un ensemble de paires clé/valeur) dont les valeurs peuvent être d'autres dictionnaires ou des représentations des valeurs de plusieurs objets. Ceci permet de définir des propriétés complexes.

4 Constitution d'une application

Une application étant un package, elle peut être reconnue comme une seule entité par le système, qui peut proposer de l'exécuter. Cela cache toute la complexité contenue dans

^⑧ Le format le plus ancien, celui utilisé dans NeXTSTEP mais aussi aujourd'hui dans GNUstep, avec de légères modifications, est un format textuel ressemblant à un mélange entre JSON et le format INI.

ce répertoire, qui est aussi un bundle, donc qui abrite une hiérarchie bien définie. Une application est un bundle dit *moderne*. Ce type de bundle se retrouve également ailleurs⁹ et est caractérisé par la présence d'un seul répertoire intitulé `Contents` à la racine, et d'un fichier intitulé `Info.plist`¹⁰ présent dans le répertoire `Contents`. Le reste dépend du type de bundle, mais il est commun, sur Mac OS X, de retrouver le répertoire `MacOS` qui abrite l'exécutable lui-même¹¹, et le répertoire `Resources` qui stocke diverses ressources (fichiers d'interface, de traduction, images, sons...).

SECTION
3

Principaux frameworks

L'existence et l'identité de l'API Cocoa, outre les petites particularités que je viens de citer, est assurée par au moins deux frameworks qu'utilise la majorité des développeurs pour Mac OS X. Toutes les classes provenant des frameworks *officiels* sont préfixées des initiales `NS`¹².

1 Foundation Kit

Le framework **Foundation** est très lié à Objective-C et à son moteur d'exécution. Il est comparable à une bibliothèque standard couvrant beaucoup d'utilisations, fournissant des types de base comme les chaînes de caractère Unicode `NSString` et les tableaux `NSArray`, mais également des classes spécifiques comme les classes permettant la sérialisation dérivées de `NSCoder`... La raison pour laquelle le framework est lié au moteur d'exécution est qu'il fournit également une classe de base pour la hiérarchie des classes, `NSObject`. Cette dernière contient beaucoup de méthodes permettant la réflexivité et joue le même rôle que la classe de base `Object` dans Java.

2 Application Kit

Le framework **AppKit** regroupe un ensemble de classes pour le développement d'applications et d'interfaces homme-machine. On y retrouve une organisation conforme à l'architecture Modèle-Vue-Contrôleur, cette dernière étant fortement encouragée pour le

⁹ Les extensions du noyau, d'extension `.kext`, qui sont des sortes de pilotes de périphérique, sont également des bundles modernes.

¹⁰ Ce fichier contient des informations importantes sur la classe principale de l'application, les formats de fichier associés, la version actuelle, l'icône ou l'affichage de l'application dans le Dock et les éléments de menu (ou *menu extras*).

¹¹ Cet exécutable est du même type que tous les exécutables UNIX qu'on peut trouver dans le système, comme `ls` ou `cat`. Il est donc possible d'y faire appel depuis la ligne de commande.

¹² Ces initiales sont celles de NextStep. La tradition les a laissées là. La sagesse populaire recommande de préfixer ses entités afin d'éviter les conflits de noms, qui sont difficiles à repérer avec Objective-C en raison de la nature dynamique du langage et du système d'envoi de messages entre objets. C'est une convention destinée à pallier l'absence d'espaces de noms.

développement d'applications Mac OS X, et de multiples classes couvrant la plupart des cas d'utilisation pour une application de bureau.

SECTION

4

Processus type de développement rapide

1 Outils de développement

Les outils de développement officiels sont fournis avec le système d'exploitation et peuvent être installés à la demande. L'élément central est l'environnement de développement intégré **Xcode** couplé à l'utilitaire de développement d'interfaces **Interface Builder**. Ce couple est issu des versions NeXTSTEP nommées **Project Builder** et **Interface Builder**¹³, et si Xcode a beaucoup évolué depuis, il n'en reste pas moins que la création d'applications se fait sur les mêmes bases.

2 Définition des classes

Il est tout d'abord primordial de créer un projet depuis un modèle d'application Cocoa puis de définir les classes du modèle et les contrôleurs de l'application. Seule l'interface de ces classes est importante à ce niveau, et la logique sera écrite plus tard.

3 Création de l'interface graphique

La première étape suivant la création du projet sous Xcode est la création de l'interface graphique. Les développeurs Mac OS X ont une relation différente avec l'éditeur d'interface graphique que ceux des autres plateformes. En effet, les éditeurs d'interface graphique sont souvent des outils qui génèrent du code lourd, peu modifiable et difficile à lire. De plus, l'ambivalence de la représentation, tantôt structurée, tantôt sous forme de code source, fait que la moindre modification isolée de l'une des deux représentations casse tout l'ensemble. Ici¹⁴, Interface Builder permet de construire une interface, qui a une représentation unique¹⁵ et qui symbolise l'ensemble des vues qui seront instanciées par AppKit de manière transparente, et aussi l'ensemble des connexions à faire avec le programme.

4 Mise en place des bindings

Une fois l'interface créée et les propriétés des éléments des vues fixées, les connexions avec le reste des classes du programme peut être fait. Interface Builder permet encore une

¹³ Avec GNUstep, on parle de **Project Builder** et de **Gorm**.

¹⁴ Une démarche similaire est disponible depuis quelques années avec GtkBuilder pour l'API GTK+, et une multitude de programmes manipulant la même représentation, dont le représentant le plus connu est Glade.

¹⁵ La représentation est stockée dans un package d'extension nib ou xib.

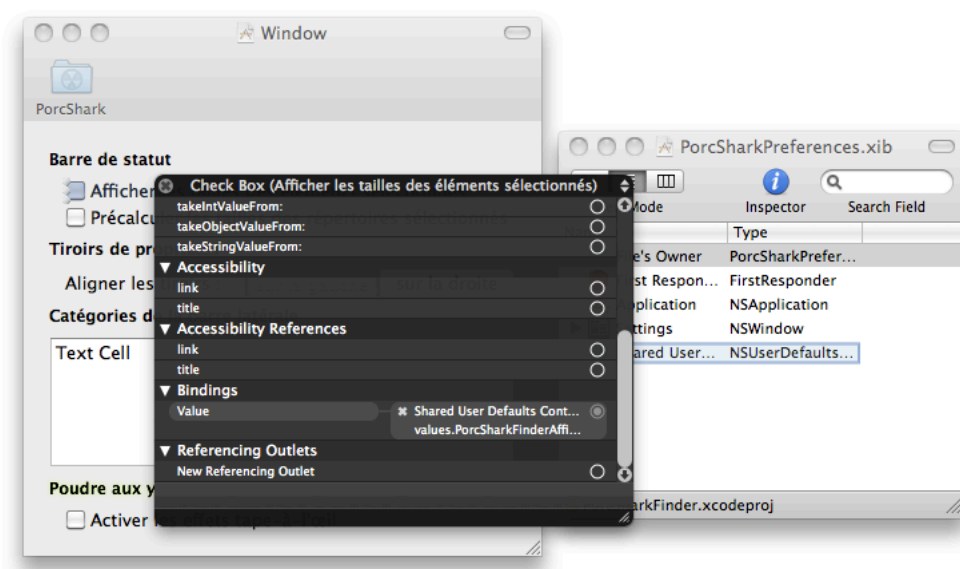


Figure 4.1. Mise en place de *bindings* dans une vue

fois d'assurer graphiquement ce travail en proposant de mettre en place des *bindings*. Il s'agit de lier des classes à des vues de l'interface, le plus souvent en définissant des *outlets*, c'est-à-dire des règles qui disent d'associer un objet représentant une vue particulière à une variable d'instance d'un objet désigné. Des comportements peuvent également être déclenchés depuis les vues (par exemple, la réponse au clic sur un bouton), en donnant une *cible* (instance d'une classe destinée à recevoir un message) et une *action* (sélecteur d'une méthode particulière qui sera appelée grâce au message). Enfin, un *binding* peut lier une propriété particulière d'une vue à une propriété de n'importe quel contrôleur ou source de données.

5 Écriture des contrôleurs

Le reste du travail concernant la logique de l'application peut être achevé dans Xcode par l'écriture des contrôleurs. À ce stade, l'application est déjà fonctionnelle puisqu'une grande partie du travail est réalisée automatiquement.

6 Le cas des délégués

La délégation est un mécanisme extrêmement courant dans l'API Cocoa. Il s'agit pour une classe d'utiliser un objet délégué qui va implémenter certaines méthodes définies dans un protocole informel et qui aura donc pour mission, du point de vue de la délégation, de conseiller l'objet déléguant, de lui fournir des informations ou de lui avertir de certaines choses. Ce qui fait qu'on n'utilise pas ce mécanisme dans les autres langages est que la

classe de l'objet délégué n'est pas forcée d'implémenter les méthodes du délégué, et ce grâce au fait qu'il est possible d'envoyer des messages à un objet même si la méthode choisie n'est pas implémentée. Les trois cas les plus courants sont les suivants.

Lorsqu'une vue est responsable de l'affichage d'un contrôle graphique de type liste, elle va proposer à l'objet qui le désire de s'inscrire comme délégué en tant que source de données. L'objet ainsi inscrit pourra proposer deux méthodes, par exemple `-(int)liste :(id)laListe nombreElements` pour donner le nombre d'éléments que la liste doit afficher, et `-(id)liste :(id)laListe elementALaPosition :(int)laPosition` pour donner un élément particulier.

Le second cas donne typiquement le rôle de conseiller à l'objet délégué. Par exemple, une classe responsable de la gestion d'une fenêtre pourra employer un objet délégué et lui demander si la fenêtre est autorisée à être fermée grâce à une méthode `-(BOOL)fenetrePeutEtreFermee`.

Enfin, le dernier cas consiste simplement à avertir l'objet délégué que certaines opérations se sont commencées, terminées, ou qu'elles vont commencer ou se terminer. On trouve un exemple de ce cas avec la classe `NSSpeechSynthesizer` qui permet l'utilisation du moteur de synthèse vocale de Mac OS X et qui peut avertir son délégué de la fin de la diction d'une phrase par le synthétiseur vocal en lui envoyant un message par la méthode `-(void)speechSynthesizer :didFinishSpeaking :` **AHCB** .

Outils pour la *modification-singe*

Le terme de modification-singe est issu d'une longue série de déformations étymologiques qui a pour origine le terme anglais de *guerilla patching*, décrivant le fait de changer le code d'un programme nommé cible à l'exécution sans se soucier des conséquences, dans les langages dynamiques. Le terme est devenu *gorilla patching* puis *monkey patching*, qui donne l'équivalent du terme que j'utiliserai ici. Cette approche induit une démarche de rétro-ingénierie. Je parlerai du programme en cours de développement comme la modification-singe.

La modification-singe exploite les caractéristiques d'un langage à typage dynamique (pour les objets et à typage faible comme Objective-C. Ceci est loin d'être une saine pratique et il en émerge des problématiques embarrassantes comme la validité des prédicats attendus par le programme cible ou la prédictibilité de la cohabitation de plusieurs modifications-singes simultanées en espace et en temps. C'est donc une voie qui doit généralement être évitée, mais dans quelques cas comme celui du projet, elle est inéluctable.

SECTION



Techniques de modification du programme à l'exécution

Objective-C recèle un nombre important de fonctionnalités obscures liées à son caractère dynamique **SOHF** dont certaines qui sont favorables à la modification-singe. Il existe un petit nombre de documents portant sur les techniques de modification-singe **MAQA** et les détails techniques à leur sujet sont assez rares. On en retrouve toutefois dans des documents traitant de la sécurité informatique de Mac OS X où sont exposées encore plus d'étrangetés **SORT**.

1 Catégories

Le développeur familier d'Objective-C pensera tout de suite aux catégories. Les catégories permettent d'étendre les classes existantes en leur adjoignant de nouvelles méthodes. Elles sont souvent utilisées pour découper l'interface et l'implémentation d'une classe en plusieurs fichiers, et plus rarement pour étendre des classes existantes pour la durée de

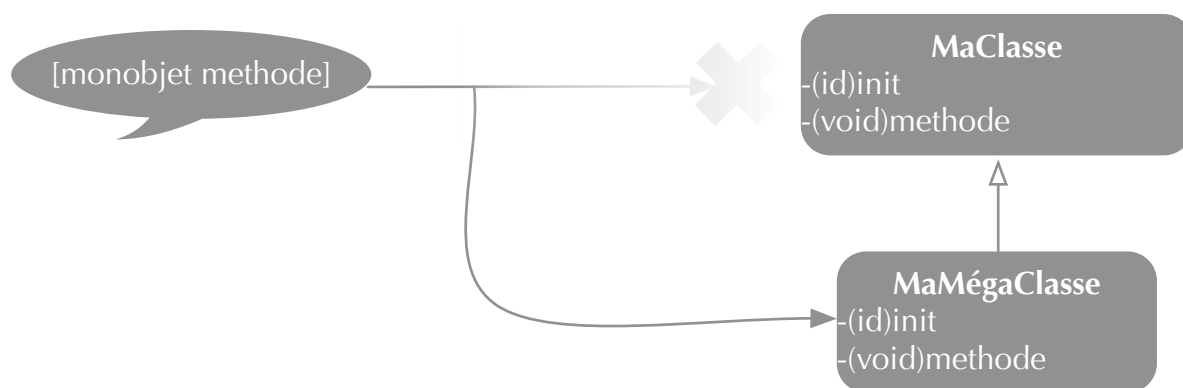


Figure 1.1. Substitution de classe

vie d'un programme. L'exemple le plus courant est l'extension de la classe `NSString` par une méthode permettant d'obtenir la chaîne de caractères inverse ^❶.

Il est possible d'offrir aux classes existantes du programme cible des méthodes permettant de réaliser des opérations commodes ou de prendre en charge des opérations sur des classes nouvelles apportées par la modification-singe.

❷ Surcharge de méthode

Une autre solution simple et sans histoire consisterait à proposer une sous-classe d'une classe donnée où les méthodes voulues auraient été surchargées. Cependant, cela nécessite de contrôler les instanciations qui sont faites, ce qui est rarement possible dans une situation de modification-singe.

❸ Substitution de classe

Le terme *class posing* n'a pas réellement de traduction française acceptée, et j'appellerai cette technique la substitution de classe. Il s'agit de remplacer dynamiquement une classe par une autre, c'est-à-dire que la classe substituante recevra les messages initialement destinés à la classe substituée. Une classe peut en substituer une autre uniquement si elle en hérite, directement ou non. Il n'est toutefois pas permis de définir de nouvelles variables d'instance, ni de recevoir les messages envoyés à la classe substituée avant l'opération de substitution. La substitution de classe permet de rajouter de nouvelles méthodes, comme les catégories, mais elle permet également de surcharger des méthodes existantes, tout en ayant accès aux méthodes surchargées : il est donc possible de modifier le comportement d'un programme de manière convenable et qui peut être invisible (par exemple, pour collecter des statistiques). La figure 1.1 donne un exemple d'utilisation.

La substitution de classe était une fonctionnalité du langage jusqu'à Objective-C 2.0, coïncidant avec la sortie de Mac OS 10.5 *Leopard*. À partir de là, elle fut considérée

^❶ Celle qui, concaténée à la chaîne originale, forme un palindrome.

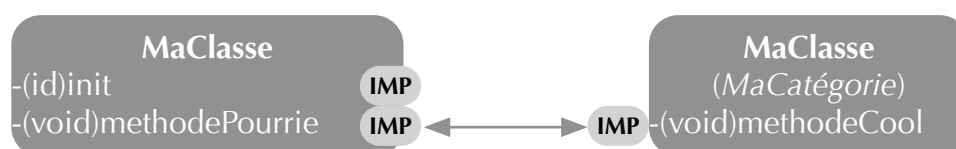


Figure 1.2. Filoutage de méthode

comme dépréciée et est indisponible en mode 64 bits. Il est donc déconseillé d'utiliser la substitution de classe.

4 Filoutage de méthode

Le terme anglais de *method swizzling* est le seul utilisé à ce jour pour décrire le crime qui consiste à agir sur l'environnement d'exécution d'Objective-C pour échanger les sélecteurs de deux méthodes dans une classe donnée. L'astuce consiste donc à écrire une catégorie d'une superclasse de la classe cible², d'y implémenter les méthodes qu'on veut filouter en ayant recours à un autre nom mais aux mêmes types dans les intrants et extrants, et d'échanger les sélecteurs à l'exécution **TCRM**.

De cette manière, après filoutage, lors de l'envoi du message [`monInstanceDeMaClasse methodePourrie`], ce n'est pas l'instance de la classe `MaClasse` du programme cible qui recevra le message mais la catégorie créée dans la modification-singe, qui présentera une autre méthode, de façon à ce que le message réellement envoyé soit [`monInstanceDeMaClasse methodeCool`]. La figure 1.2 illustre cette situation.

Il n'y aurait aucun avantage à utiliser cette technique si on ne pouvait pas avoir recours à la méthode précédemment filoutée dans la méthode qui filoute. À l'exécution, il faut penser que les sélecteurs auront été échangés. Ainsi, le corps de la méthode `-(void)ModificationSinge_aboyer` ne devra pas appeler [`self aboyer`] car cela provoquerait un appel récursif. Il faudra bien utiliser [`self ModificationSinge_aboyer`] pour appeler la méthode originale.

Depuis Objective-C 2.0, le nombre de lignes de code pour réaliser cette opération a considérablement diminué (il en faut maintenant trois : deux pour identifier les attributs des deux méthodes, et une pour réaliser l'échange) et a éliminé tous les cas particuliers gênants qui pouvaient être rencontrés avant. Même en sachant cela, j'ai préféré opter pour l'utilisation de `JRSwizzle`³, une petite classe réalisant l'opération de manière correcte et sécurisée avec n'importe quelle version d'Objective-C. De cette manière, l'opération est réalisée avec des sécurités supplémentaires en cas d'échec du filoutage.

² La raison est qu'il est nécessaire de pouvoir accéder à une classe connue pour en faire une catégorie.

³ Voir <http://github.com/rentzsch/jrswizzle>.



Figure 1.3. Exemple concret de boxage de canard

5 Implémentation superséquent

Un article de 2008 expose les limites du filoutage de méthode et propose une alternative **SIXC** connue sous le nom anglais de *supersequent implementation*. La principale objection au filoutage concerne la localité de la méthode filoutée d'une classe par rapport à sa hiérarchie : on ne peut pas filouter la méthode `dealloc` de `NSSuperWindow`, classe dérivée de `NSWindow`, si cette méthode n'est pas implémentée (surchargée) dans `NSSuperWindow`, car le filoutage ne fonctionnera pas. On est dans l'obligation, dans ce cas, de devoir filouter toutes les `NSWindow`. Et même dans le cas où on peut s'en sortir, tout simplement en testant dans la méthode filoutée la classe à laquelle appartient l'objet courant `self`, il existe un risque qu'après mise à jour du programme cible, si la hiérarchie accueille une classe fille, disons `NSMegaWindow`, qui est utilisée en lieu et place de `NSSuperWindow` et qui surcharge la méthode `dealloc`, le filoutage ne fonctionne plus.

L'auteur propose donc une manière de surcharger une classe tout en laissant la liberté d'appeler l'implémentation originale, mais en évinçant tous les inconvénients du filoutage. Il y a évidemment des inconvénients, dont la lenteur à l'exécution⁴ et l'impossibilité à utiliser dans un nombre limité de cas, comme le cas où on fait appel à des classes utilisant la redirection de messages. L'utilisation raisonnée de cette technique dans les cas limites peut être toutefois envisagée.

6 Boxage de canard

Le boxage de canard, ou *duck punching*, est un des noms de la modification-singe, mais symbolise plus précisément le fait d'utiliser la propriété de typage canard d'un langage

⁴ En effet, il faut parcourir manuellement la liste des implémentations pour un sélecteur donné dans toute la hiérarchie, et s'arrêter à la prochaine implémentation trouvée et valide. L'exécution auparavant en temps constant grâce au *cache* des méthodes se fait maintenant en temps linéaire, en fonction du nombre de méthodes et de la hiérarchie des classes.

pour parvenir à introduire des instances d'une classe dont on ne connaît que l'interface (ou du moins une partie) dans un système préexistant. L'idée est que si un canard ne fait pas le bruit que l'on veut, il n'y a qu'à lui donner des coups de poing jusqu'à ce qu'il fasse le bon bruit **RC07**. La figure 1.3 donne un exemple de boxage de canard.

Avec Objective-C, cela permettrait de résoudre un problème très particulier. Soit une classe du programme cible inaccessible à la compilation (et à l'édition de liens) mais dont il faudrait pouvoir tirer des instances pour les passer à un autre objet du programme cible. À moins de disposer d'une fabrique dans le programme cible, il faut donc être en possession des entêtes de la classe voulue et d'au moins quelque chose de concret à donner à l'éditeur de liens pour la compilation si l'on veut instancier cette classe. Une implémentation *similaire* d'une classe de même nom et de même interface ne fonctionne pas, car l'environnement d'exécution d'Objective-C ne peut alors déterminer quelle classe utiliser puisque les deux ont le même nom. Le typage canard permettra tout de même de fournir un objet de type différent mais de même interface au programme cible, puisque dans certains cas le type n'est pas observé mais qu'il est juste nécessaire que l'interface soit implémentée.

7 Utilisation de l'écosystème de classes du programme cible

Grâce à la dynamique de Objective-C, il est possible d'arriver à utiliser de manière satisfaisante l'écosystème de classes du programme cible depuis un autre programme, en invoquant des classes non connues à la compilation. Les liens se font à l'exécution, de manière tardive. On peut donc parvenir à appeler les méthodes d'une classe donnée grâce à la macrocommande `NSClassFromString()`, ce qui permet par exemple d'instancier des objets d'une classe dont on n'a pas la définition, de stocker cette instance dans un pointeur d'objet `id`, puis d'envoyer des messages à cette instance.

On peut même aller plus loin puisqu'il est possible de briser l'encapsulation d'une classe donnée du programme cible de manière à exposer une de ses variables d'instance. Ceci est particulièrement utile lorsqu'on veut naviguer parmi des associations de manière plus rapide, en exposant une nouvelle méthode accesseur dans une catégorie de la classe ciblée, ou tout simplement pour accéder à une variable d'instance afin de lui faire subir un traitement, depuis une catégorie. Cela permet de contourner le fait que l'accès direct aux variables d'instance nécessite de connaître la définition de la classe. On utilise donc la fonction `object_getInstanceVariable()` qui remplit un pointeur fourni avec un pointeur vers la variable d'instance demandée.



Exploration statique d'un exécutable Mac OS X

Cette section présente quelques outils spécialisés que j'ai pu utiliser afin d'extraire des informations sur le comportement d'un programme depuis le fichier exécutable du pro-

gramme lui-même. Il s'agit d'une connaissance *statique* qui me renseignerait sur l'architecture du programme cible et son comportement prédéfini.

❶ Récupération de données sur les classes

Il existe plusieurs utilitaires permettant de récupérer des données sur une classe, les plus évidents étant les outils UNIX standard `nm` et `strings` qui permettent de dévoiler des noms d'entités ou des chaînes de caractères ayant un rapport avec le programme. L'utilitaire le plus avancé disponible pour les exécutables Cocoa se nomme `class-dump`⁵ et permet, entre autres choses, d'extraire des informations sur les classes d'un programme et de les ranger sous la forme très commode de fichiers d'en-tête Objective-C. Ceci est possible du fait que les exécutables Objective-C possèdent le savoir de leur propre architecture en eux, afin d'assurer la réflexivité du langage.

❷ Lecture du code machine annoté

Une manière d'obtenir des informations sur le comportement d'un exécutable Objective-C est d'utiliser un outil, présent dans les outils officiels du développeur, nommé `otool`, ou un frontal graphique pour cet outil comme `otx`⁶. `otool` permet d'extraire des informations sur les formats de fichier binaires compris par Mac OS X, et en particulier le code machine contenu dans ces fichiers. La lecture de ce code est abrupte serait quasiment inutile si `otool` n'essayait pas de l'annoter de façon à marquer chaque envoi de message de façon lisible (c'est-à-dire écrit comme on écrirait un envoi de message), et en séparant le code correspondant à chaque méthode. Ceci permet d'avoir un aperçu rapide du comportement de méthodes que l'on voudrait modifier, afin de pouvoir influencer le moins possible sur le reste du programme cible, en simulant correctement leur comportement.

Malheureusement, si `Finder.app` est devenu plus facile à lire depuis Mac OS 10.6, en raison de sa transition de Carbon vers Cocoa, il n'en reste pas moins une application *signée*⁷ dont `otool` ne parvient pas à annoter le code. Il est possible toutefois de décrypter les binaires signés en utilisant le débogueur `gdb` afin de récupérer des parties de la mémoire du programme qui ont été décryptées par le système de signature, puis de replacer ces parties de mémoire dans une copie de l'exécutable d'origine **REMX** qui pourra alors être soumis à un examen par `otool`, par exemple. Cependant, mes essais n'ont pas été fructueux sur le Finder.

⁵ Voir <http://www.codethecode.com/projects/class-dump/>

⁶ Voir <http://otx.osxninja.com/>

⁷ La signature du code est un mécanisme de protection apparu avec Mac OS 10.5 et qui a le double but d'éviter la falsification de programmes importants par des virus, et également de garantir que le programme s'exécute sur du matériel Apple (ce dernier but n'est jamais réalisé en pratique, à cause d'une grande communauté dédiée à l'installation de Mac OS X sur du matériel tiers).

3 Extraction d'informations depuis les packages d'interface

Les packages d'interface sont inclus dans les bundles d'application. Ceci est fort utile pour AppKit qui va pouvoir faire tout le travail magique pour faire apparaître des vues déjà liées aux classes et aux attributs, mais on pourrait croire que c'est également utile pour regarder la manière dont est développée une application ou une fenêtre particulière, surtout au niveau des *bindings* qui ont été définis. Malheureusement, la réalité fait que la plupart des packages d'interface `.nib` ou `.xib` sont *compilés* avant d'être inclus dans les applications : il est donc impossible de les ouvrir avec Interface Builder. Toutefois, une parade est possible ⁸ mais elle ne permet pas de réutiliser le package d'interface ainsi lu.

SECTION

3

Exploration dynamique d'un exécutable Mac OS X

Cette section présente les outils qu'il est possible d'utiliser pour étudier la structure et le comportement d'un exécutable Mac OS X durant son exécution. C'est aussi l'occasion de présenter l'outil que j'utilise pour expérimenter des modifications sur le programme en temps réel.

1 Un langage pour communiquer avec Cocoa

Il existe un langage de script destiné aux développeurs Cocoa qui permet d'interagir avec les objets Cocoa. **F-Script**⁸ est dérivé de Smalltalk et sa compatibilité totale avec Cocoa fait qu'il est très utilisé pour le débogage ou pour offrir un langage afin d'étendre un programme existant. F-Script fournit une interface graphique modulaire sous la forme d'un navigateur d'objets⁹ et d'une fenêtre permettant d'utiliser directement l'interpréteur.

2 Utilisation d'un débogueur

Il est possible d'utiliser la version officielle d'Apple du débogueur GNU `gdb`, dont Xcode propose une interface graphique intégrée, pour attacher le débogueur à un processus existant. La commande `attach` de `gdb` permet à GDB de s'infiltrer dans un processus. Ensuite, il est possible de poser des points d'arrêt sur des symboles donnés afin d'observer pas-à-pas le déroulement du programme.

Il est également possible d'envoyer des messages Objective-C à des classes ou à des objets. Ceci peut permettre de charger des bundles dynamiquement dans le programme en cours d'exécution, puis d'utiliser les classes du bundle nouvellement chargé. C'est de cette façon qu'il est possible d'injecter F-Script dans n'importe quelle application Cocoa. En effet, F-Script propose un framework permettant d'utiliser le langage et son interface

⁸ Voir <http://www.fscript.org>.

⁹ Ce navigateur rappelle ceux qui sont disponibles dans certaines implémentations de Smalltalk.

graphique dans une nouvelle application. Ce qu'on fait est donc de charger dynamiquement ce framework depuis `gdb`, puis de lancer l'interface graphique de F-Script, ou bien de rajouter au menu de l'application un menu pour F-Script.

La distribution officielle de F-Script propose un service Automator qui, une fois lancé, permet d'injecter F-Script dans l'application au premier plan, automatisant ainsi, c'est le cas de le dire, la tâche décrite au paragraphe précédent.

3 Utilisation de l'interface F-Script pour l'exploration

L'injection du menu F-Script dans une application n'est pas toujours aisée. Par exemple, `Dock.app` n'expose ni menu principal, ni icône d'application – qui serait à afficher dans lui-même, ni élément de barre de menu. De plus, son statut d'application particulière fait que certaines classes courantes ne sont pas présentes et cela déstabilise F-Script. Dans le cas de `Finder.app`, la procédure habituelle s'applique, et un des auteurs de F-Script présente même quelques copies d'écran montrant l'exploration de ce programme avec F-Script **UPFC**.

SECTION

4

Création d'un bundle compatible SIMBL

L'étude technique réalisée précédemment a mis en évidence le système SIMBL **10** qui permet de charger dynamiquement dans des applications ciblées des bundles précis. SIMBL est une couche désirable entre les mécanismes de bas niveau permettant le chargement dynamique et les bundles à charger car il joue un rôle de médiation et permet un ciblage précis.

En réalité, SIMBL est chargé avec toutes les applications Cocoa mais il ne fait rien en lui-même à part tenir une liste des bundles qu'il peut charger en fonction des applications supportées, et charger les bundles au moment opportun.

1 Créer et placer le bundle

Concevoir un bundle compatible avec SIMBL revient à créer un simple bundle sous Xcode **CCRE**. Les informations utiles à SIMBL seront placées dans le fichier `Info.plist`, dans le dictionnaire `SIMBLTargetApplications`, qui permettra de définir les identifiants et les plages de versions des applications ciblées.

Lors du chargement du bundle par SIMBL, la méthode statique `+(void)load` de la classe principale est appelée afin de donner un point d'entrée au bundle créé. L'appel est effectué lorsque l'application cible est déjà initialisée. C'est donc à partir de ce point qu'il est possible de créer des nouveaux objets, de filouter quelques méthodes, d'installer des observateurs...

10 Voir <http://www.culater.net/software/SIMBL/SIMBL.php>

Une fois le bundle compilé, il suffit de le placer dans un des emplacements que SIMBL surveille au démarrage d'une application. Un de ces emplacements est `~/Library/Application Support/SIMBL/Plugins/`.

② Faciliter le développement du bundle

Il est utile de mêler les phases de développement et d'exploration : en l'absence d'une version totalement faite en F-Script du bundle, il est utile de pouvoir tester régulièrement le bundle, mais il est également utile de pouvoir continuer à explorer et à expérimenter avec les classes du programme lorsque le bundle est injecté, afin de tester les cas conflictuels ou pour les besoins du débogage.

L'inclusion du framework F-Script dans le bundle en cours de création est donc possible, et cela permet, au chargement du bundle, de charger également l'interface graphique de F-Script, afin de disposer automatiquement du bundle en cours de création et du système F-Script. Les zones d'influence des deux entités sur les classes du programme cible sont disjointes, il n'existe donc pas de conflit au chargement.

SECTION 5

Éthique

S'il est tentant de vouloir opérer des milliards de petites modifications sur les classes existantes, il faut néanmoins raisonner l'utilisation de la modification-singe au strict nécessaire.

Chaque modification dans une méthode existante met en doute la validité des prédicats qu'elle respectait alors, et le bris de certaines conditions peut mener à des résultats embarrassants.

① Conséquences du bris de certaines conditions

Le bris d'une condition sur la valeur de retour indique que la méthode ne fait pas au moins la même chose que ce qu'elle faisait avant (réduction sémantique), ce qui peut avoir des répercussions négatives sur le programme, incluant un comportement indéterminé, le plantage par non-vérification d'une assertion, ou la modification inattendue du flot d'exécution due à la survenue d'une exception. Il est simple, en général, de localiser ces conditions puisque le sélecteur de la méthode est en général assez verbeux pour décrire ce qu'elle fait.

Le bris d'une condition sur les effets de bord indique que l'un des cas rajoutés à l'ensemble des cas que traite la méthode n'a pas été complètement défini. Les conséquences peuvent être diverses, mais par exemple si l'appelant s'attend au remplissage d'un pointeur qui était nul auparavant, le programme peut planter et les causes du plantages peuvent être très difficiles à évaluer, puisqu'il aurait lieu après l'invocation de la méthode défectueuse.

De plus, il peut être difficile de déterminer quels sont les effets de bord d'une méthode s'ils sont multiples.

Le bris d'une condition sur le temps d'exécution indique que la méthode effectue des opérations qu'elle devrait réaliser ailleurs. En effet, certaines méthodes sont prévues pour être appelées très souvent et donc pour être exécutées de façon rapide, et une modification-singe mal placée peut rendre lourd un programme qui était très léger auparavant. Une des seules choses qu'il est possible de faire, mais qui ne couvre pas tous les cas, est d'évaluer le contexte d'utilisation de la méthode du point de vue de la fréquence d'utilisation, en associant à l'appel de la méthode originelle une entrée dans le *log*, ou bien la production d'un bip système, et en quantifiant la fréquence d'utilisation à partir de ces données ponctuelles et visibles.

Le bris d'une condition sur la réentrance¹¹ est encore plus vicieux et très difficile à déterminer. Il faut pouvoir connaître le contexte d'utilisation de la méthode et évaluer selon chaque cas.

2 Attitude à adopter pour modifier

Je ne connais pas de protocole permettant d'être efficace pour déterminer l'ensemble des conditions que doit respecter une méthode externe donnée, et c'est pourquoi je pense qu'on peut agir par essais et erreurs, en gardant en tête les cas évoqués ci-dessus qui sont le fruit de mon expérience sur ce projet.

3 Survie dans le temps

Une modification-singe est tributaire de l'organisation interne du programme qu'elle modifie. Si une partie du programme cible concernée par la modification vient à changer, par exemple à cause d'une mise à jour, alors le changement peut concerner la structure, c'est-à-dire que des classes ont changé, n'existent plus ou sont remplacées, ou bien il peut concerner le comportement, c'est-à-dire que certaines conditions sur les méthodes ne sont plus celles qui ont été établies auparavant. Dans tous les cas, ces changements sont dommageables pour la modification-singe, qui doit en conséquence être adaptée. Cet effort constant d'adaptation au fil du temps doit être le prix à payer pour le développement d'une modification-signe robuste et fiable, ce qui est bien plus difficile à réaliser que le développement d'une application possédant les mêmes qualités.

SECTION

6

Conclusion

Ce projet avait été proposé avec l'intention de me munir de l'arsenal que j'avais alors suggéré, n'ayant pour seul indice de faisabilité que l'existence de programmes utilisant des techniques similaires. Toutefois, l'ensemble des outils à la disposition du développeur

¹¹ À prendre au sens du terme anglophone de *thread-safety*.

a été envisagé, et ce n'est qu'après l'avoir vérifié que j'ai pu avoir la certitude qu'aucune autre solution plus déterministe ne pouvait être employée pour les fins du projet. Le plus simple serait que Apple publie une interface de programmation d'extensions officielle.

Réalisation

Ce chapitre discute de l'organisation et de la réalisation des fonctionnalités. Le bundle est divisé en plusieurs classes regroupées selon la fonctionnalité qu'elles contribuent à réaliser. Il existe donc sept groupes :

- le groupe des **classes principales** qui régissent le fonctionnement de l'ensemble et proposent le point d'entrée du programme *via* les méthodes statiques `+(void)load` et `+(void)install` de la classe `PorcSharkMain` ;
- le groupe du **method swizzling**, c'est-à-dire le filoutage de méthodes, où se trouve la classe `JRSwizzle` qui a été intégrée au projet ;
- le groupe des **préférences de l'application**, qui ont pour but de gérer les préférences de l'utilisateur concernant le programme, et également l'interface permettant de les modifier ;
- le groupe des **classes communes** (ne pas confondre avec les classes principales), qui rassemble les structures de données communément utilisées ainsi que tout ce qui a un impact sur plus d'un groupe ;
- le groupe des **trucs tape-à-l'œil**, qui rassemblait toutes les classes inutiles qui ont pour unique but de faire dire ``*Ouah !*'' aux spectateurs de la présentation ^❶ ;
- le groupe de la **barre de status** qui permet de réaliser la fonctionnalité d'affichage de la taille des éléments sélectionnés ;
- le groupe de la **barre latérale** qui permet de réaliser la fonctionnalité de regroupement des emplacements de la barre latérale en espaces de travail ;
- le groupe du **tiroir de propriétés** qui permet de réaliser la fonctionnalité d'affichage des tiroirs de propriétés omniprésents.

Ce chapitre donne les détails d'implémentation sur ces groupes. Y seront donc présentées les classes incriminées, une description sommaire de ce que font les méthodes à filouter, ce qui a été fait et ce qui n'a pas pu être fait.

Concernant le filoutage, je présumerai que l'attitude par défaut consiste à étendre le champ d'action des méthodes filoutées (par exemple, rajouter un élément dans un tableau qui sera renvoyé, ou rajouter un nouveau cas correspondant à une fonctionnalité nouvelle) plutôt qu'à remplacer le comportement existant. De cette manière j'ai une ga-

❶ Ce groupe inutile est présent dans le code mais n'est pas activé. Nous n'en parlerons pas, mais un coup d'œil dans le code est possible pour les personnes intéressées.

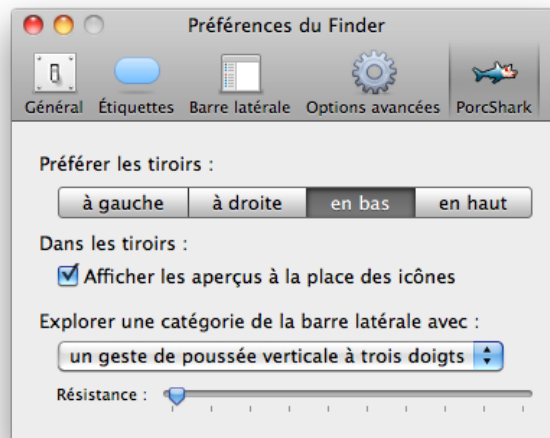


Figure 1.1. Un nouveau venu dans les préférences du Finder

rantie supplémentaire que le nouveau code s'exécutera bien comme l'ancien dans tous les cas existants, mais que le nouveau code couvre les nouveaux cas induits par la modification.

SECTION



Préférences d'application

Pour faire fonctionner les préférences du projet, j'ai dû tout d'abord comprendre le système des préférences d'application, placer les lectures et écritures de ces préférences au bon endroit, puis faire enregistrer un ensemble de valeurs par défaut. J'ai dû également décortiquer la façon dont fonctionne la fenêtre des préférences du Finder pour lui adjoindre un nouveau panneau abritant toutes ces préférences. Enfin, j'ai ajouté un sous-menu au menu Présentation pour activer ou désactiver une des préférences.

1 Utilisation et mise en place des préférences

a Le système de préférences des applications Mac OS X

Chaque application peut stocker ses préférences, souvent appelées *defaults*, de manière automatique et standardisée. Chaque fichier de préférence est placé dans le répertoire `~/Library/Preferences/` sous un nom correspondant à l'identificateur du bundle²

² L'identificateur d'une application est écrit sous la forme d'un *reverse DNS*, comme par exemple `com.apple.Finder` ou bien `name.oin.PorcSharkFinder`.

suffixé de l'extension `.plist` des fichiers de listes de propriétés.

b Lecture et écriture des préférences avec Cocoa

La classe `NSUserDefaults` de Foundation Kit propose d'accéder aux préférences d'une application simplement et de manière automatique sous forme d'une instance acquise via le message `[NSUserDefaults standardUserDefaults]`. L'écriture et la lecture des préférences se fait comme pour un dictionnaire grâce à des méthodes comme `-(void)setObject:forKey:` et `-(id)objectForKey:` : cette partie est la plus simple car Cocoa fait tout le travail.

c Mise en place de préférences par défaut

Lors du démarrage du programme, il faudrait spécifier des valeurs par défaut dans le cas où aucune préférence n'a jamais été choisie par l'utilisateur.

La mise en place des préférences par défaut se fait en remplissant un nouveau dictionnaire modifiable, de type `NSMutableDictionary`, et de le passer au dictionnaire des préférences de l'application en lui envoyant le message `-(void)registerDefaults:`.

2 Interface graphique de choix des préférences

Si jusque là la tâche fut facile, c'est parce qu'il n'y avait aucune différence avec la manière de procéder dans une application Cocoa. Maintenant, la tâche est beaucoup plus ardue puisqu'il s'agit de concevoir l'interface de choix des préférences et de l'intégrer dans la fenêtre des préférences du Finder. J'ai préféré ajouter un panneau de préférences plutôt que d'intégrer les différents éléments d'interface dans les panneaux existants, de manière à ce que l'interface ne soit pas cassée en cas de mise à jour de ce panneau par Apple³.

a Conception de l'interface

Avec Interface Builder, j'ai créé un nouveau package d'interface, que j'ai nommé `PorcSharkPreferences.xib`.

Les panneaux de préférences des applications Mac OS X sont en général des fenêtres standard munies d'une barre d'outils et de différentes vues que l'on change à l'aide de la barre d'outils.

J'ai donc créé une fenêtre contenant une barre d'outils. J'ai affecté un seul bouton à cette barre d'outils : il s'agit du bouton qui va être incorporé à la barre d'outils existante du Finder un peu plus tard. Puisque je ne rajouterai qu'une vue de préférences, ou qu'un panneau, je peux insérer mes composants à même la *content view* de ma fenêtre⁴. Les états des cases à cocher et du contrôle à choix présents dans la fenêtre sont directement reliés aux préférences de l'application par des *bindings* que j'ai choisis.

³ On peut imaginer quelque chose de simple comme l'ajout d'une case à cocher dans les préférences générales, qui pourrait être fait dans la prochaine mise à jour de Mac OS X.

⁴ Chaque fenêtre peut être munie d'une barre d'outils, d'une vue racine qui la couvre entièrement (*root view*) et d'une vue principale contenant ce qui est en-dessous de la barre d'outils (*content view*).

La *content view* et le bouton de barre d'outils ont été reliés à la classe `PorcSharkPreferences` responsable de la gestion du panneau de préférences et de son inclusion dans la fenêtre du Finder.

b Squattage du panneau dans la fenêtre du Finder

Dans la fenêtre de préférences d'une application classique, les vues sont regroupées dans un *Tab view*, qui est une vue jouant le rôle de conteneur donnant l'accès à des onglets pour changer de vue enfant. Ici, le *Tab view* n'affichera pas ces onglets. Cependant, on garde tout de même quelques avantages à procéder de cette façon, puisqu'en demandant au *Tab view* de changer de vue, elle essaiera de se redimensionner pour s'adapter à son contenu. C'est ainsi que font plusieurs applications, et j'ai voulu savoir si le Finder n'échappait pas à la règle.

Visor⁵ est un bundle SIMBL qui s'attaque à l'application `Terminal.app` pour proposer l'affichage d'une fenêtre de terminal à la façon des panneaux de console dans les jeux vidéo de tir à la première personne, comme `Quake III Arena`⁶. Son code source est ouvert et il m'a permis d'étudier la façon dont un panneau est rajouté dans les préférences. Son fonctionnement répond à la généralité décrite au paragraphe précédent.

Toutefois ce n'est pas le cas pour le Finder qui fonctionne différemment. Une analyse grâce à F-Script a permis de constater que toutes les vues susceptibles d'être affichées sont des enfants de la *content view* de la fenêtre des préférences et que le passage d'une vue à l'autre est assurée par une classe contrôleur nommée `TPreferencesWindowController`, qui délègue à d'autres contrôleurs dérivés de `TPaneController` les opérations d'*ouverture* et de *fermeture* des panneaux dont ils ont la responsabilité. Il y a ainsi un type de contrôleur par panneau, et c'est la méthode `-(id)controllerForPaneAtIndex` : qui renvoie l'instance du contrôleur approprié qui va contrôler l'affichage de la bonne vue et la disparition des autres.

Afin de faire entrer mon panneau dans la danse, j'ai été dans l'obligation de boxer un canard pour créer une classe contrôleur pouvant se faire passer pour une classe fille de `TPaneController`, en implémentant les mêmes méthodes⁷. Une fois mon animal traumatisé, j'ai pu alors filouter la méthode `-(id)controllerForPaneAtIndex` : pour la remplacer par une méthode de mon cru qui étendait son travail afin de prendre en compte le cas où l'indice demandé correspond à celui de mon panneau de préférences.

Avant de m'éloigner du champ lexical des mauvaises actions, j'ai dû accomplir une autre tâche qui concernait l'ajout du bouton de barre d'outils défini dans mon package d'interface droit dans la barre d'outils de la fenêtre de préférences du Finder. J'ai été forcé de découvrir le fonctionnement des barres d'outils dans Cocoa en me cassant les dents à maintes reprises. En réalité, un élément de type `NSToolbarItem`, peut représenter une facette particulière d'un élément de barre d'outils selon l'état de la barre d'outils (afficher

⁵ Voir <http://github.com/darwin/visor>.

⁶ Visor est d'ailleurs le nom d'un des modèles de personnages qu'il est possible d'incarner dans ce jeu.

⁷ Ces méthodes fournissent comme accesseurs un pointeur vers la fenêtre de préférences et la vue à afficher, ainsi que deux méthodes `-(void)open` et `-(void)close` qui sont responsables de l'affichage et de la disparition du panneau, comme précisé plus haut.

uniquement le texte, ou bien uniquement les icônes, ou bien les deux) et l'universalité d'une barre d'outils, qui peut être partagée à travers plusieurs fenêtres⁸. Ce qui différencie les différents éléments de barre d'outils n'est donc pas la nature des instances de `NSToolBarItem` mais les identifiants assignés à ces *items*. Un identifiant est une valeur, comme une chaîne de caractères ou un nombre entier. J'ai donc pris soin de définir l'identifiant `'PorcShark'` pour le bouton de barre d'outils à rajouter à la barre d'outils de la fenêtre des préférences du Finder. La barre d'outils est symbolisée par la classe `NSToolbar` et elle possède un délégué à qui elle peut demander quels sont les identifiants qu'il est acceptable de sélectionner (méthode `-(id)toolbarSelectableItemIdentifiers` :) et informer que l'*item* d'un identifiant donné va être ajouté à une barre d'outils, laissant au délégué la possibilité de fournir un *item* particulier (méthode `-(id)toolbar:itemForIdentifier:willBeInsertedIntoToolbar` :). Le délégué de la barre d'outils de la fenêtre des préférences du Finder est défini comme étant l'instance actuelle du contrôleur de la fenêtre de préférences. Ainsi l'insertion du nouvel élément de barre d'outils passe par le filoutage des deux méthodes déléguées ci-dessus. La méthode `-(void)configureToolbar` de ce contrôleur est celle qui s'occupe de la mise en place de la barre d'outils de la fenêtre des préférences du Finder (elle est assez spéciale car elle ne propose aucune option de configuration à l'utilisateur). Il convient de la filouter également car c'est à ce stade qu'il faut demander à la barre d'outils d'insérer un *item* ayant pour identifiant `PorcShark` à la dernière position libre.

SECTION

2

Pause pipi : gestion des fichiers dans le Finder

Cette section a pour but de présenter brièvement les différentes représentations des fichiers dans le Finder. Je parle ici des représentations internes, à savoir des structures C, des classes Objective-C ou même des classes C++. En effet, il existe différentes manières de représenter les fichiers, et découvrir puis gérer toutes ces représentations a été difficile. Il est apparu que les représentations pouvaient être classées en familles. Cette section clarifiera les choses en présentant plusieurs familles ayant des utilités diverses.

1 Famille FENode

La structure C `OpaqueNodeRef` est un type opaque associé à ce que nous appellerons un nœud, et qui peut être un élément du système de fichiers ayant une réalité physique, ou bien un élément totalement virtuel mais destiné à servir de conteneur et à être navigué, comme les éléments *Ordinateur* ou *Réseau* dans le Finder.

La structure C `TFENode` référence un élément de type `OpaqueNodeRef`. Elle est utilisée par une multitude de classes pour référencer un nœud de manière directe.

⁸ On rencontre ce cas dans l'application Mail.app : même si on crée différentes fenêtres pour rédiger un nouveau message, lorsqu'on enlève ou rajoute un élément à la barre d'outils, le changement se répercute dans toutes ces fenêtres.

Enfin, la structure C `TFENodeVector` décrit une liste linéaire de nœuds en référençant trois éléments de type `TFENode` par le biais d'un idiome utilisé couramment par le Finder pour d'autres éléments, et par la STL de C++ pour implémenter le type `std::vector`. Le premier élément pointe sur le début de la liste, le second sur la fin de la liste, et le troisième pointe sur le dernier emplacement alloué couramment⁹. La raison de la présence d'une telle chose est que le Finder a quelques classes écrites avec Objective-C++¹⁰, ce qu'on remarque dans les *class dumps* par la présence des méthodes `-(id).cxx_construct` et `-(void).cxx_destruct` qui sont appelées à la construction et la destruction des objets Objective-C pour lancer respectivement la construction et la destruction de leurs variables d'instances étant des objets C++. Ici donc, on présume que les classes Objective-C demandant des pointeurs de `TFENodeVector` utiliseront en réalité le type `std::vector<TFENode>`, ce qui leur permettrait de naviguer à travers la liste de nœuds facilement. Je n'ai pas cherché à exploiter C++ pour ce projet, pensant que cela allait m'amener à une impasse due au fait que C++ possède un typage statique et qu'il serait plutôt impossible de travailler sur des classes C++ dont on ne connaît pas la définition.

2 Famille FNode

Le Finder conserve plusieurs informations sur les nœuds, à savoir les propriétés des fichiers qu'ils représentent, leurs icônes... Peu importe que le type `OpaqueNodeRef` stocke ces informations ou qu'elles soient stockées autre part, il est possible de récupérer des informations sur un nœud `TFENode` en instanciant la classe `FNode`. Une méthode de classe est disponible pour obtenir un objet `FNode` correspondant au `OpaqueNodeRef` pointé en paramètre. Cela permet d'accéder aux propriétés d'un objet, qui sont récupérées par le Finder d'une manière qui nous importe peu, et qui peuvent changer au cours du temps en fonction de ce que peut récupérer le Finder probablement dans un autre fil d'exécution¹¹.

Ainsi `FNode` est la classe privilégiée pour récupérer des informations sur un fichier, et c'est une des classes que j'utiliserai à ces fins. Il est à noter que le Finder définit une catégorie sur la classe `NSArray`, représentant un tableau dans Foundation Kit, et offrant une méthode de classe permettant d'obtenir la liste des `FNode` correspondant à un `TFENodeVector`.

3 Famille NSNavNode

Il existe une multitude de classes de préfixe `NSNav` qui sont utilisées par l'ensemble du système, de manière indirecte, pour accéder aux facilités que le Finder propose, et notamment pour générer les feuilles d'ouverture ou de sauvegarde de fichiers. Elles font quelque

⁹ Les conteneurs de la STL ont une stratégie de réservation de mémoire que l'on peut paramétrer grâce à la classe `std::allocator` et que l'on peut utiliser grâce à la fonction membre `reserve`.

¹⁰ Objective-C++ est un mélange des deux langages de façon à ce qu'un programme puisse utiliser les deux sans que les classes Objective-C n'interfèrent avec les classes C++, mais les unes peuvent utiliser les autres.

¹¹ La conception de cette partie n'est pas connue et il n'est pas nécessaire de la connaître.

peu double emploi avec les classes du Finder puisqu'elles sont légion et implémentent chaque aspect présent dans les feuilles d'ouvertures, des différentes vues de navigation à la barre latérale, elle aussi dédoublée. On peut voir une trace des classes qui m'intéressent dans de nombreux rapports de plantage d'applications Mac OS X mais il n'existe aucune trace de leur utilisation par des développeurs¹². La classe de base du système de représentation des nœuds utilisé ici est `NSNavNode`, qui est dérivée en `NSNavFBENode`, représentant un nœud (physique ou virtuel) référencé par un `OpaqueNodeRef`, et `NSNavVirtualNode` représentant un nœud non rattaché à un `OpaqueNodeRef` et dont les caractéristiques peuvent être construites comme on le souhaite¹³. On peut remarquer trois sous-classes de `NSNavFBENode` pour les trois cas particuliers suivants :

NSNavFBENodeQueryNode pour les résultats de recherche de Spotlight¹⁴ ;

NSNavFBENetworkNode pour l'élément *Réseau* ;

NSNavFBETopLevelNode pour l'élément *Ordinateur* (ou ``placez-le-nom-de-votre-ordinateur-ici").

On peut obtenir des renseignements divers d'un `NSNavNode` comme on peut le faire avec un `FINode`, mais là où cette classe est plus intéressante, c'est qu'il est possible d'obtenir un élément de type `NSNavFBENode` à partir d'un URL ou d'une chaîne de caractères représentant un chemin complet.

SECTION

3

Affichage de la taille des éléments sélectionnés

La classe `PorcSharkStatusBar` est le point central qui coordonne la mise à disposition de la fonctionnalité d'affichage de la taille des éléments sélectionnés. Cela demande de connaître deux choses importantes :

- d'une part, l'endroit où afficher le résultat, et la manière d'y parvenir ;
- d'autre part, la manière de récupérer l'information à afficher.

¹² Ma démarche pourrait en ce sens être prise comme une avancée de la connaissance et une victoire de plus dans le combat qui oppose les développeurs d'applications Mac OS X aux ingénieurs pommés de la Silicon Valley !

¹³ On peut donc penser que l'utilisation de cette classe permet de faire des *faux* fichiers en spécifiant toutes les caractéristiques du faux fichier, y compris parent et enfants. On peut donc construire des hiérarchies totalement imaginaires et les représenter dans le Finder ou dans les feuilles d'ouverture et d'enregistrement, et c'est ce qui est fait par exemple pour les media (provenant d'iTunes, iPhoto ou iMovie) qu'il est parfois possible d'explorer depuis une feuille d'ouverture ou d'enregistrement. On peut imaginer beaucoup d'applications intéressantes à ceci, par exemple pour constituer une hiérarchie de fichiers basée sur des étiquettes et non pas des répertoires.

¹⁴ Je ne prétendrai pas ici expliquer l'intégralité du fonctionnement de Spotlight, mais il existe une librairie pour utiliser le démon UNIX qui y est rattaché et qui est responsable de l'indexation, ainsi qu'un utilitaire en ligne de commande du nom de `mdutil`, ce dernier pouvant être très pratique pour utiliser Spotlight en ligne de commande.

valeur	texte en français
5	Recherche sur ...
6	éléments, Go disponibles
7	Go disponibles
8	éléments (ou sur sélectionnés)

Table 6.1. Différents types de texte pouvant être affichés

Quelques octets, 4 sur 22 sélectionnés, 54,05 Go disponibles

Figure 3.1. Un petit texte préfixant les indications de la barre de statut

1 Affichage d'un texte

La partie qui concerne cette fonctionnalité se nomme *barre de statut*. Elle est présente sous deux formes dans le Finder, dépendamment du mode dans lequel la fenêtre se trouve actuellement (mode navigation ou mode spatial). L'important est donc de faire une modification qui puisse toucher les deux formes. L'observation de cette zone de l'interface montre que la barre de statut concerne non seulement l'affichage d'un champ de texte qui sera l'objet de la modification voulue, mais également une série de contrôles divers dont des images représentant le mode d'affichage des fichiers ou l'état d'une éventuelle synchronisation sur disque distant.

La classe responsable de la génération du texte à modifier est `TStatusTextFldController`, et c'est sa méthode `-(NSString*)statusTextForType :outWidth :` qui doit être filoutée pour contrôler le texte à afficher. Le premier argument de cette méthode est un nombre entier désignant quel type de texte doit être affiché. Mes observations ont permis de mettre en évidence les cas présentés dans le tableau 6.1, mais les autres cas m'échappent puisqu'ils doivent probablement correspondre à des situations auxquelles je n'ai pas accès ¹⁵.

Le deuxième paramètre de la méthode à filouter est un pointeur vers un nombre de type `double` qui doit accueillir la largeur du texte qui sera affiché. La classe `NSString` de Foundation est détachée de toutes les considérations graphiques possibles puisque c'est la classe de base pour gérer les chaînes de caractères, mais il existe heureusement une classe `NSAttributedString` provenant de `UIKit` et permettant de représenter du texte enrichi et d'en extraire certaines informations. C'est de cette façon que je peux remplir correctement le pointeur passé en paramètre et ainsi respecter les postconditions de la méthode filoutée.

¹⁵ La synchronisation du disque `iDisk` est un bon exemple.

2 Formatage du résultat et internationalisation

À dater de la présentation, ce qui était présenté était presque terminé, et le formatage du résultat était réalisé par une méthode d'instance qui calculait un ordre de grandeur pertinent et renvoyait une chaîne de caractères comportant la valeur dans l'ordre de grandeur pertinent avec la bonne unité, en français. Depuis, j'ai réussi à mettre en évidence la classe `TFileSizeFormatter` du Finder, dérivée de `NSNumberFormatter`, qui permet de donner des représentations en chaîne de caractère d'une taille, en respectant les contraintes d'internationalisation et de choix d'ordre de grandeur en vigueur. J'ai donc pu utiliser ce type de formateur directement ¹⁶ et m'assurer que ce qui est affiché est cohérent avec le reste des informations qui sont délivrées à l'utilisateur par le Finder.

3 Récupération de la taille des éléments sélectionnés

La récupération de la taille des éléments sélectionnés passe par l'extraction de tous les éléments présents dans la vue, puis dans le filtrage des éléments sélectionnés uniquement, ou alors de tous les éléments lorsqu'on n'a pas de sélection courante.

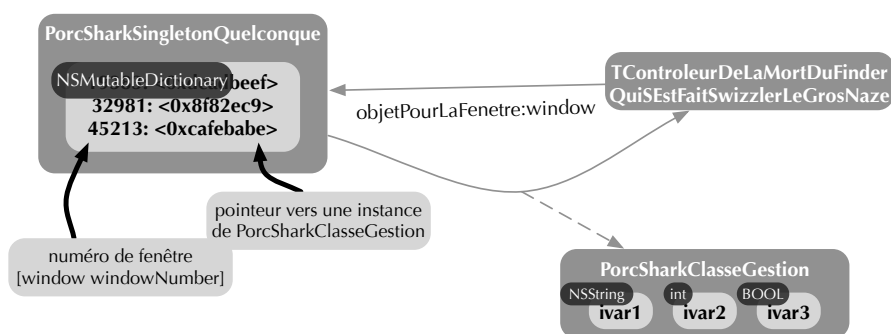
La vue affichant les fichiers du répertoire courant est symbolisée par une classe précise. Dans le mode de vue par icônes, la vue utilisée est de type `TIconView`. N'importe laquelle de ces vues descend de la classe `IKImageBrowserView` dont les instances possèdent un objet délégué répondant au protocole informel `TBrowserViewDataSource`, qui va donner ce qu'il faut afficher. Heureusement il est possible de trouver un objet de type `TBrowserViewController` qui peut répondre à ce protocole et renvoyer chaque élément, donc chaque fichier ou répertoire, ou chaque élément sélectionné.

Maintenant que les représentations des nœuds sont connues, on est en mesure d'obtenir pour chaque élément de la vue du navigateur une instance de `FINode` le représentant et conservant des informations le concernant. Les accesseurs intéressants pour la classe `FINode` sont `-(long long)fileSize` et `-(long long)fileSizeSync` qui vont renvoyer respectivement la taille du fichier (pris comme un fichier UNIX) et la taille du fichier (qui peut être un fichier conventionnel ou un package).

Lorsque le fichier est un répertoire, il convient de s'en informer grâce à la méthode `-(BOOL)isContainer`. Le mécanisme de calcul de la taille d'un répertoire n'est pas encore bien déterminé, mais a peut-être un rapport avec la méthode `-(id)size :(BOOL)` qui renvoie la chaîne de caractères représentant la taille du fichier avec l'unité de mesure adéquate. Dans tous les cas, lorsque la taille d'un répertoire est déterminée, elle peut être donnée grâce à la méthode `-(long long)fileSize`. Si cette taille n'est pas encore connue, alors la méthode renverra un nombre négatif.

Je n'ai pas pu mettre en lumière ce qui permet d'engendrer le calcul de la taille d'un répertoire donné dans le Finder. La diversité des représentations et le fait qu'il est impossible de savoir ce qui se cache derrière certaines représentations opaques en est la cause. Toutefois, il est à rappeler que la plupart des navigateurs affichent la taille des éléments qui ne sont pas des répertoires par défaut, et qu'ils affichent uniquement la taille des fi-

¹⁶ Ici, j'ai fait une liaison tardive avec la bonne classe grâce à `NSStringFromClass()`.

Figure 4.1. L'idiome *fake ivar* en images

chiers dans la sélection lorsque sélection il y a. Cette décision est explicable en prenant l'exemple d'un répertoire comme le répertoire racine, dont le calcul de la taille peut demander un temps considérable. D'autres stratégies existent à mi-chemin, et consistent à automatiser le calcul de la taille d'un répertoire lorsqu'il semble qu'il n'est pas trop volumineux¹⁷. Pour le livrable final, cette partie de la fonctionnalité est donc abandonnée. En contrepartie, lorsque le Finder aura calculé la taille d'un répertoire après une demande implicite de calcul (ouverture d'une fenêtre d'informations, dans la plupart des cas), le bundle saura afficher cette nouvelle information lorsque le répertoire est sélectionné à nouveau. De plus, lorsqu'un des éléments sélectionnés est un répertoire dont la taille est inconnue, le bundle affiche au moins la taille connue de toute la sélection en spécifiant qu'en réalité elle est plus grande.

SECTION

4

Pause ca...fé : ajout de variables d'instance à des classes à l'exécution

Cette section introduit un idiome que j'ai utilisé à plusieurs reprises pour pallier une des limites du concept de catégories avec Objective-C. Le problème est qu'il est impossible de rajouter des variables d'instance à une classe depuis une catégorie. Or il est souvent nécessaire de garder des états supplémentaires introduits par les modifications. La solution est alors d'utiliser un singleton qui peut gérer un dictionnaire associant un objet à une valeur donnée. Le plus souvent, la valeur sera l'identifiant numérique d'une fenêtre, de manière à associer des objets à chaque fenêtre. La figure 4.1 présente les interactions entre les classes en présence, à savoir celle de la méthode filoutée qui a fait naître le besoin, celle du singleton qui gère les associations et celle des objets associés aux fenêtres.

¹⁷ Évidemment, on ne peut pas décider du calcul de la taille d'un répertoire en fonction de sa taille, mais des indices faciles à obtenir comme le nombre d'éléments qu'il contient ou le nombre d'éléments cumulés sur un petit nombre de niveaux de sa sous-hiérarchie peuvent être examinés.

Regroupement des emplacements en espaces de travail

La barre latérale du Finder n'est présente dans les fenêtres qu'en mode navigation. La vue qui affiche le contenu de cette barre latérale est dérivée d'un composant particulier de AppKit géré par la classe `NSOutlineView`. L'utilisation de cette vue est assez complexe, même pour développer une application normale. Cette complexité est due au fait qu'elle est capable d'afficher une hiérarchie d'éléments ayant des représentations potentiellement différentes.

1 Nature et comportement de la barre latérale

La barre latérale est un composant particulier dans le Finder. En effet, la liste des éléments qu'elle affiche est partagée dans tout le système, puisqu'on retrouve cette barre latérale dans les feuilles d'ouverture ou d'enregistrement de fichier dans les applications, ainsi que dans toutes les fenêtres du Finder. La modification d'un élément de cette liste a des répercussions sur l'ensemble des barres latérales ouvertes. La barre latérale peut accepter des éléments par drag and drop, les éléments peuvent être retirés également par drag and drop, et chaque élément donne accès à un menu contextuel bien précis. De plus, certains éléments affichent un bouton, par exemple un bouton d'éjection de périphérique ou d'image disque. La barre latérale présente quatre catégories fixes, qui sont les appareils locaux, les emplacements partagés accessibles sur le réseau, les emplacements et les modèles de recherche. Ces catégories peuvent être affichées ou cachées mais pas renommées ou déplacées. Il n'y a pas de réorganisation possible.

2 Enregistrement des informations de la barre latérale

Les informations de la barre latérale sont enregistrées par le Finder dans un fichier de préférences situé à `~/Library/Preferences/com.apple.sidebarlists.plist`. Il contient quatre éléments pour les quatre zones à afficher, et chacun de ces quatre éléments contient des propriétés diverses ainsi qu'une liste d'éléments. Chaque élément est constitué de trois informations : le titre de l'élément, l'icône au format base64, et l'alias au format base64. L'alias est une forme codée permettant de référencer un nœud sur le disque. Il est possible d'en tirer un URL pour l'exploiter dans un programme `SPUA`.

Il est toutefois difficile de savoir quand le Finder accède à ce fichier, et l'écriture concurrente sur ce fichier peut engendrer sa corruption. La méthode la plus sécuritaire consisterait à modifier la représentation de la barre latérale au sein du Finder, mais ceci n'est guère évident.

3 Problèmes relatifs à la barre latérale

La classe `NSOutlineView` présente un délégué agissant comme source de données. Cela consiste en plusieurs fonctions qui vont aider à déterminer les enfants d'un élément,

si l'élément doit être affiché comme un groupe... La classe `TSidebarViewController` est destinée à remplir le rôle de source de données, et plusieurs méthodes annexes permettent de fournir des informations sur le contenu de la barre latérale. C'est là qu'il était question de décider entre modifier le comportement de l'intégralité des méthodes de source de données, et donc agir à un niveau proche de l'affichage, ou alors comprendre le modèle qui sous-tend le comportement de ces méthodes et modifier ce modèle.

La classe `TSidebarViewController` permet de dévoiler la notion de **zone** utile à la compréhension du fonctionnement de la barre latérale. Une zone est une catégorie parmi les quatre fixes citées plus haut. Quelques méthodes permettent d'obtenir des informations sur ces zones et de savoir à quelle zone appartient une cellule donnée. Il existe également des méthodes permettant de construire une liste d'éléments graphiques de la liste reliés à une zone donnée.

La barre latérale est l'objet de plusieurs représentations dont je n'ai pas pu déterminer le dénominateur commun. En effet, outre les classes `TSidebarViewController`, `TSidebarView`, `TSidebarItemCell` et autres, qui servent à représenter la barre latérale dans les fenêtres de navigation, il existe des classes responsables de la gestion de la barre latérale dans les préférences du Finder ¹⁸, ainsi que des classes responsables de son affichage et de sa gestion dans les feuilles d'ouverture et de fermeture, comme `NSNavSidebarController`, qui en plus présentent quelques particularités ¹⁹.

L'étude complète de cette fonctionnalité est trop vaste pour le temps qui m'était alloué. D'un côté, mes essais en modifiant la barre latérale au plus près de la vue grâce aux méthodes déléguées n'ont pas été concluants dans le sens où il était alors nécessaire de faire interférer deux modèles différents (le modèle lié aux éléments réellement présents dans la barre latérale, et celui des éléments rajoutés artificiellement) et que cela engendrait trop de problèmes. De l'autre côté, le fait que j'ai manqué d'informations sur ce que fait le code et quels messages sont envoyés m'a empêché de trouver la source sur laquelle se synchronisent toutes les représentations de la barre latérale. De plus, cela aurait demandé un nombre très conséquent de modifications puisque chaque représentation tient pour acquis qu'il n'existe que quatre zones de base.

Je pense que cette modification est faisable dans les conditions où elle était présentée, mais que sa réalisation nécessite une étude approfondie des classes et des éléments externes en jeu.

4 Lot de consolation

Néanmoins, j'ai essayé de proposer une réponse différente au problème posé en donnant accès rapidement à un autre niveau de hiérarchie pour les éléments de la barre latérale. Ainsi je partirai du principe qu'on veut souvent accéder à des sous-répertoires de ceux qui

¹⁸ Ce panneau de préférences est par ailleurs un peu dérangent dans la mesure où la barre latérale est représentée de manière fixe, ce qui va à l'encontre de la possibilité de l'amélioration unifiée de la barre latérale.

¹⁹ Il s'agit notamment d'une fonctionnalité méconnue permettant d'enregistrer des recherches dans la barre latérale de façon à ce qu'elles soient disponibles dans une seule application, ainsi que de la zone nommée **DONNÉES** permettant d'accéder aux media des applications iLife.

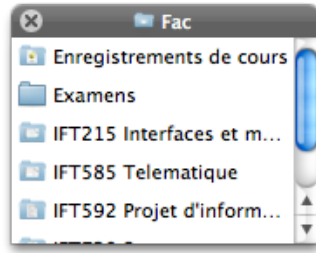


Figure 5.1. Le lot de consolation

sont déjà présents dans la barre latérale. Ma proposition consiste donc à offrir l'accès à ces sous-répertoires de manière très rapide.

a Description

La barre latérale reste sous la forme qu'on connaît mais les fenêtres de navigation affublées de cette barre latérale possèdent une fonctionnalité non envahissante qu'il est possible d'activer avec un trackpad multitouch²⁰ seulement. La fonctionnalité pourrait être à l'avenir être activée par d'autres moyens. Il est possible de choisir le geste qui active la fonctionnalité parmi :

- l'écartement de deux doigts (*zoom out*),
- la rotation de deux doigts,
- ou le balayage vertical de trois doigts (*swipe*)

Il est également possible de choisir la sensibilité du mouvement sur une échelle restreinte, dans les préférences. Le geste d'activation consiste donc à effectuer le mouvement multitouch en ayant la souris positionnée sur un élément de la barre latérale. Une fois ce geste réalisé, une petite fenêtre similaire à celle de la figure 5.1 apparaît sous la souris et présente une liste des sous-répertoires de l'élément de la barre latérale choisi. La réalisation du geste multitouch inverse ferme la fenêtre, tandis que le clic sur un des sous-répertoires amène la fenêtre de navigation à ce point.

5 Réalisation

Chaque fenêtre de navigation est associée à une instance de la classe `PorcSharkSidebarHandler`. C'est le singleton `PorcSharkSidebar` qui gère ces associations à la manière de l'idiome *fake ivars*. Les méthodes filoutées permettront uniquement de détecter un geste d'activation de la petite fenêtre, et c'est l'objet `PorcSharkSidebarHandler` correspondant qui doit alors s'occuper de la petite fenêtre, exposant une méthode permettant son activation pour une référence de nœud `TFENode` et un point de l'écran donnés.

²⁰ Snow Leopard semble supporter le multitouch pour les ordinateurs sortis après mi-2008.

a Détection des gestes multitouch

Les gestes multitouch font l'objet d'événements envoyés aux vues, qui y répondent ou non en implémentant ou non une méthode déléguée au nom évocateur. Il existe plusieurs façons de prendre en charge les gestes multitouch, l'une assez automatisée permettant de détecter les *zooms*, rotations et balayages à trois doigts, et l'autre plus complexe permettant d'obtenir le maximum d'informations du trackpad et de suivre chaque étape du mouvement. Il n'a pas été nécessaire d'en arriver là, puisque j'ai pu implémenter les méthodes déléguées suivantes dans les vues concernées :

Zoom `magnifyWithEvent :(NSEvent*)event` en prenant en compte le paramètre [event magnification]

Rotation `rotateWithEvent :(NSEvent*)event` en prenant en compte le paramètre [event rotation]

Balayage vertical `swipeWithEvent :(NSEvent*)event` en prenant en compte le paramètre [event deltaY]

J'ai été contraint d'utiliser le filoutage pour implémenter ces méthodes uniquement pour la classe `TSidebarView` qui gère la vue de la barre latérale, mais j'ai pu les implémenter simplement dans une sous-classe de `NSPanel` ²¹ qui est `PorcSharkTiroirPanel`.

La sensibilité à ces gestes multitouch est calculée en fonction du paramètre de résistance qu'il est possible de choisir dans les préférences. Je pense qu'il est possible de la calibrer mieux que ce qu'elle n'est calibrée actuellement.

Il est important de signaler que la réception de ces événements multitouch dépend de l'activation de ces gestes dans le panneau de préférences Trackpad des Préférences Système.

b Gestion de la liste des sous-répertoires

La classe `PorcSharkSidebarHandler` gère l'affichage de la petite fenêtre et se présente comme source de données pour la liste de type `NSTableView` qui accueillera les répertoires. Plusieurs méthodes sont donc implémentées pour donner le nombre d'éléments de cette liste et chaque élément. Cette classe se présente également comme délégué de la liste de type `NSTableView`, ce qui permet donc de réagir à la sélection d'un élément de la liste par l'invocation de la méthode `retargetToNodeAndComputePath` : sur le contrôleur de fenêtre de navigation courant, mais également de personnaliser l'affichage de chaque élément de la liste. Pour cela, des instances de la classe `TSidebarItemCell` sont créées et affectées à chaque élément de la liste des sous-répertoires.

Cette liste est recrée lors d'une invocation de la méthode `-(void)apparaître :aLaPosition :`, qui va explorer les sous-répertoires d'un `TFENode` donné grâce à l'instance de `NSNavFBENode` correspondante et notamment sa méthode `getNodeAsInfoNode` :

²¹ `NSPanel` est une classe dérivée de `NSWindow` qui représente une petite fenêtre du type inspecteur, ou une fenêtre noire transparente *style HUD* comme c'est le cas ici. Il est même possible d'instancier directement cette sous-classe depuis Interface Builder. J'ai rajouté des effets graphiques lors de l'apparition ou la disparition de cete fenêtre, mais je les ai désactivés pour que l'affichage paraisse fluide.

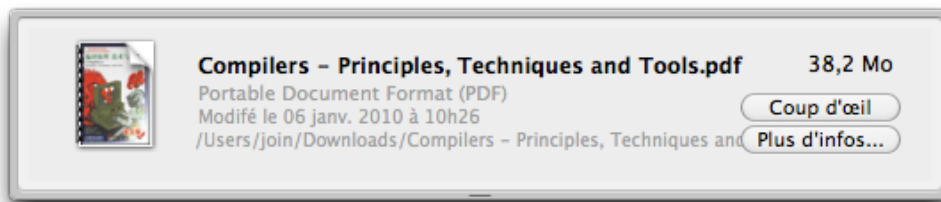


Figure 6.1. Un tiroir de propriétés horizontal qui touche à mon livre de chevet

qui permet de récupérer une version explorable de cette instance. Ensuite, il aura suffi de filtrer les divers éléments du tableau retourné par la méthode `children` pour ne garder que les répertoires, puis de les stocker dans un tableau gardé comme variable d'instance dans la classe `PorcSharkSidebarHandler`. Les méthodes déléguées en tant que source de données utilisent alors ces informations pour fournir à la vue `NSTableView` ses éléments.

SECTION

6

Tiroir de propriétés omniprésent

Les fenêtres de propriétés et les inspecteurs de propriétés n'ont pas le même comportement : les fenêtres de propriétés sont attachées à un fichier précis, tandis que les inspecteurs de propriétés sont attachés à la sélection courante ou au répertoire courant dans le cas où il n'y a pas de sélection. Dans tous les cas, on remarque que les mêmes vues sont utilisées et quelles sont séparées en petites sections qu'on peut afficher ou cacher, sauf la section d'en-tête montrant entre autres l'icône, la taille et le titre de ce qui est inspecté. Je vais dans un premier temps passer en revue ces aspects puis peser le pour et le contre de leur utilisation.

On peut même aller plus loin dans l'application de l'architecture MVC en proposant une vue complètement nouvelle pour afficher les mêmes informations : ainsi la vue est totalement adaptée au contexte d'utilisation voulu. C'est l'approche qui a été adoptée et le détail de sa réalisation sera abordé dans un second temps.

1 Fonctionnement des fenêtres et inspecteurs de propriétés

Les fenêtres de propriétés sont gérées par la classe `TInfoWindowController` (ou `TInspectorWindowController` dans le cas de l'inspecteur) qui dirige une fenêtre de propriétés ou un inspecteur et qui contrôle les différentes sections, que le vocabulaire des développeurs du Finder semble appeler *slices*. Chaque *slice* est en réalité une instance de la classe `TSliceView` et la vue qui contient toutes les *slices* est de type `TSliceContainerView`. La gestion de ce qu'affichent les sous-vues des différentes *slices* est la responsabilité de différents contrôleurs dont il est possible d'avoir une liste grâce à la méthode

`-(NSMutableArray*)infoViewControllers` du contrôleur gérant une fenêtre ou l'inspecteur de propriétés. Les deux implémentent cette méthode, ce qui laisse supposer qu'il existe un protocole commun ou une classe contrôleur commune. Cette supposition est validée puisque les deux classes descendent de la classe `IInfoWindowControllerBase`. Il y a donc un contrôleur par *slice* et chaque contrôleur est chargé de gérer ce qui est affiché et la position des éléments affichés.

En explorant un peu plus loin chaque contrôleur on peut voir qu'ils descendent tous de la classe `TInfoWindowViewController`. La méthode `-(id)valueControllers` donne un aperçu de l'enfer : les éléments sont des instances de la classe `IPropertyValueController` qui gèrent pour chaque propriété un *value extractor* ou extracteur de valeur (classe `IPropertyValueExtractor`) chargé d'aller extraire une information à partir d'une référence vers un fichier, et un *layout binder* (classe `TLayoutBinder`) associé à une vue (ou *slave view*) qui sera repositionnée et redimensionnée pour afficher convenablement la valeur. Il existe plusieurs contrôleurs²² gérant chacun une propriété précise. Ce petit tour d'horizon de l'univers des fenêtres et inspecteurs des propriétés laisse possibles plusieurs solutions pour remplir les tiroirs de propriétés. Dans tous les cas, l'accès aux éléments sélectionnés est fait de la même façon que pour la fonctionnalité d'affichage de la taille des éléments sélectionnés. Il s'agit de méthodes déléguées qui sont filoutées et gérées par la classe `PorcSharkBrowserWindowHook` dans la partie commune.

J'avais envisagé de créer une sous-classe de la classe `IInfoWindowControllerBase` pour gérer un conteneur de propriétés, ou bien de créer un contrôleur personnalisé qui générerait des *slices* choisis avec soin. Cependant, j'ai dû revoir ma position pour plusieurs raisons. Tout d'abord, l'affichage des propriétés est sujet à une hiérarchie de classes spécifiques très contrôlée où le couplage est très fort. Connaître et maîtriser le comportement de toutes ces classes aurait pris beaucoup de temps, et il n'aurait pas été plus facile d'afficher tout d'un coup que d'afficher toutes les *slices* une par une. De plus, l'affichage de propriétés modifiables dans un tiroir facilement accessible est assez dangereux, et le contenu est trop gros pour s'afficher dans un tiroir qui devra être assez fin pour ne pas être gênant.

La solution la plus facile était également ici la plus satisfaisante : il s'agit de gérer le contenu du tiroir des propriétés de manière indépendante.

2 Conception de la vue du tiroir

J'ai donc créé deux nouveaux fichiers d'interface `Interface Builder` pour concevoir la vue du tiroir : l'un pour un tiroir vertical affiché quand on souhaite l'afficher à gauche ou à droite de la fenêtre, et l'autre pour un tiroir horizontal, comme à la figure 6.1, affiché quand on souhaite l'afficher en haut ou en bas de la fenêtre. Les vues de ces deux fichiers d'interface sont reliées aux mêmes *outlets* d'une classe `PorcSharkTiroirManager` dont chaque instance est associée à une fenêtre de navigation dans le singleton `PorcSharkSidebar`, dans la plus stricte tradition de l'idiome *fake ivars*. La construction d'une instance de `PorcSharkTiroirManager` entraîne la création du tiroir et le remplissage de sa vue

²² J'ai dénombré au moins 46 classes nommées `TProperty X Controller`, souvent appariées à des `TProperty X Extractor`.

principale (*content view*) par l'outlet référençant la vue principale du fichier d'interface qu'il faut, ce dernier étant chargé au tout début.

3 Attachement du tiroir des propriétés aux fenêtres

C'est la classe `TBrowserWindowController` qui est instanciée lorsque le besoin se fait ressentir de créer une nouvelle fenêtre de navigation. On pourrait donc filouter la méthode `-(id)initWithState :targetPath :` qui est utilisée comme constructeur afin de créer une nouvelle association avec une nouvelle instance de `PorcSharkTiroirManager`. Cependant, la classe `PorcSharkBrowserWindowHook` est responsable d'un filoutage bien plus sympathique, puisqu'il s'agit du filoutage de la classe `TGlobalWindowController` qui gère toutes les fenêtres dignes d'intérêt, et particulièrement les fenêtres de navigation. Il est prévenu à chaque fois qu'une fenêtre est activée, et c'est lui qui est appelé lors de la création d'une fenêtre grâce à la méthode `-(id)createWindowWithTarget :forceContainer :spawnOrigin :desiredLocation :` par laquelle passent toutes les demandes de création de fenêtre de navigation. C'était naturellement le meilleur endroit pour faire le travail d'attachement d'un nouveau tiroir de propriétés.

4 Actualisation du contenu du tiroir

Le contenu du tiroir est actualisé grâce à la méthode `actualiser` de la classe `PorcSharkTiroirManager`. Cette méthode est invoquée dans plusieurs situations, la plus courante étant lorsqu'un changement survient dans la vue de la fenêtre de navigation. Le filoutage de la méthode déléguée `viewChanged` de la classe `TBrowserWindowController` qui contrôle une fenêtre de navigation était le meilleur choix.

La récupération des éléments à décrire dans le tiroir se fait de la même manière que pour la barre de statut. Seules changent les informations qu'on tire des instances de `FINode` récupérées. Il n'y a donc rien de particulier à dire, sinon qu'on utilise une instance de la vue `TLabelSwatch` pour afficher la couleur de l'étiquette d'un fichier de la même manière que le Finder, et qu'on traite les cas particuliers de la corbeille et d'une recherche en cours.

5 Récupération de l'icône d'un fichier

Un dernier point de difficulté a été de remplir le tiroir avec l'icône du fichier sélectionné. Cette dernière peut être obtenue facilement grâce à la méthode `icon` de la classe `FINode`, mais cette icône a une aire de 128 pixels carrés seulement. Afin d'obtenir une représentation à la bonne taille de l'icône, j'ai utilisé la méthode `createImageForMaximumSize :options :` de la classe `FINode`, qui a produit une référence vers une image Core Graphics²³. Il a fallu convertir cette image en `NSImage` utilisable dans une vue de type `NSImageView` comme celle qui existe dans la vue du tiroir.

²³ C'est un format utilisé par l'API Core Graphics qui n'est pas directement utilisable en tant qu'image Cocoa, à savoir `NSImage`.

Il fallait également pouvoir proposer de récupérer l'aperçu d'un fichier donné, et cette opération était un peu plus complexe. Pour récupérer l'aperçu d'un fichier, il est possible d'utiliser l'API QuickLook qui centralise la production d'aperçus de fichiers dans tout le système. J'ai utilisé une catégorie à rajouter à la classe `NSImage` mise au point par M. Gemell afin de faciliter la création d'aperçus par QuickLook de manière bloquante. Ainsi, au début, il m'était possible d'afficher un aperçu du fichier sélectionné, au prix d'une attente parfois longue lorsque l'aperçu QuickLook n'était pas encore généré. L'interface utilisateur était totalement bloquée pendant quelques secondes.

J'ai donc utilisé la classe `NSOperationQueue` qui permet de gérer aisément les tâches d'arrière-plan, en fournissant à l'instance globale de `NSOperationQueue` que je gérais une nouvelle opération sous forme de bloc²⁴, dont la charge était de réaliser l'opération bloquante de création d'un aperçu QuickLook, puis de changer l'image de la vue du tiroir concerné pour afficher l'aperçu.

SECTION
7

Packaging d'un plugin SIMBL

Les applications Mac OS X ont la réputation d'être faciles à installer et à désinstaller. En effet, une application est un paquet autosuffisant qui peut être déplacé à loisir et sa suppression est la seule chose à faire pour désinstaller une application²⁵. Or un plugin pour SIMBL n'est pas à proprement parler une application, et SIMBL l'est encore moins. L'auteur de SIMBL propose un installateur pour son application, et autorise la redistribution de SIMBL à condition de l'accompagner de sa licence originale (GNU GPL v2). De plus l'installation de notre plugin SIMBL nécessite au moins l'installation de SIMBL et le redémarrage du Finder. Ce n'est pas une procédure d'installation triviale et c'est pourquoi la question du *packaging* méritait d'être posée.

J'ai donc été confronté à l'utilisation de l'application **Package Manager** disponible dans les outils du développeur Mac OS X et j'ai pu constater que j'aurais dû consacrer une demi-douzaine d'heures à la lecture de sa documentation. J'ai préparé le paquet permettant d'installer le plugin au bon endroit sur le système ainsi qu'un script de post-installation qui redémarre le Finder.

J'ai dû ensuite créer une image disque avec l'**Utilitaire de disque** officiel de Mac OS X, puis je l'ai personnalisée avec le Finder lui-même et mon logiciel de dessin favori. Il est courant que les applications se transportent par le biais d'une image disque dont le contenu est personnalisé afin d'indiquer les opérations à effectuer à l'utilisateur. C'est donc ce que j'ai fait puisque l'installation comporte deux étapes, à savoir installer SIMBL avec son installateur si ce n'est pas déjà fait, puis installer l'extension PorcSharkFinder.

²⁴ Les blocs sont un ajout récent au langage Objective-C et sont appelés *fermetures* ou *closures* dans d'autres langages. Il s'agit d'une manière d'inclure une fonction interne anonyme consciente des variables de la portée de l'endroit où est défini le bloc. On simule ce comportement avec des classes internes anonymes en Java, et des foncteurs en C++.

²⁵ Ce n'est pas vrai si on considère que les fichiers de préférences installés sont également à enlever.

Conclusion

Les fonctionnalités implémentées dans le Finder sont au nombre de trois, cependant le programme qui avait été fixé n'a pas été respecté en totalité. Chaque fois que l'abandon d'une partie d'une fonctionnalité a été décidé, je constate que c'est dû principalement à un manque de temps plutôt qu'à une limite technique. J'étais loin de me douter, lorsque j'ai proposé ce projet, que j'allais peiner à terminer ces trois fonctionnalités qui me semblaient triviales. Certaines parties auraient sans doute demandé le double du temps consacré à ce projet, qui a été consacré à la fois à la capitalisation d'une expérience et d'un savoir sur le sujet qui n'est pas documenté en français, et qui est peu documenté de toute manière, et à la réalisation des fonctionnalités sus-mentionnées, le second objectif nécessitant l'accomplissement d'une partie non négligeable du premier.

Si j'avais à refaire ce projet, je me mettrais probablement à la recherche de coéquipiers car il est difficile de progresser dans l'ombre lorsqu'on est seul, et il est moins facile de faire des erreurs. Je pense que le Finder n'est pas le programme le plus facile à modifier (je vais parler de ce nouveau verbe à l'Académie Française), du fait de sa place centrale dans le système, et qu'il aurait été peut-être bien plus facile de modifier un programme non signé, ou du moins une application comme Mail ou iCal. Il y a cependant un programme encore plus dur à décrypter, du fait de son statut de programme sans menu ni fenêtre, c'est le Dock. Mes quelques tentatives de m'y introduire, en début de parcours, n'ont pas été grandioses, mais l'expérience que j'ai acquis durant cette session et rassemblé dans ce rapport aurait peut-être raison de lui. J'aurais de toute manière installé un environnement de développement virtualisé, car en son absence il n'était pas très évident d'utiliser un Frankenfinder non fini, plein de bugs et sujet à des plantages aléatoires, surtout quand j'avais besoin de gérer des fichiers.

Je trouve que ce type d'activité est très enrichissant à tous points de vues, puisqu'on est confronté à l'inconnu sans cesse et la compréhension de ce qui se passe est une nécessité, que l'on peut découvrir ce qui se passe sous le capot d'un système fermé, que l'on intègre des connaissances diverses qui ne se limitent pas à un langage précis. J'ai pu apprendre à utiliser un environnement de programmation très différent de ce que j'étais habitué à utiliser jusque là, et une interface de programmation très élégante et complète que je peux maintenant utiliser à peu près comme un grand. J'ai pu également me familiariser avec des côtés plus *hardcore* comme le débogage d'un programme signé en cours d'exécution et je

CHAPITRE 7. CONCLUSION

suis maintenant capable de tirer le moindre détail d'un rapport d'erreur. Cela m'a ouvert les yeux sur plusieurs problèmes intéressants, et je peux rajouter le fait que j'ai construit quelque chose qui me sert déjà quotidiennement à cette liste de choses que le projet m'a apporté. Pour finir, le projet m'a également apporté des ennuis, car le cahier des charges que j'avais établi n'est rempli qu'en partie.

Je pense que proposer ce type de projet, soit en continuant l'existant, soit en s'attaquant à un nouveau cas, avec comme documents de base ce projet, sous forme d'activité pédagogique libre de longue durée serait une chose intéressante à l'avenir, et je m'empresserai de le proposer aux enseignants qui m'entourent.



Bibliographie

- TECH** Précédent rapport, Étude technique
- SPEC** Précédent rapport, Spécification des fonctionnalités
- PREL** Précédent rapport, Rapport préliminaire
- HIG** Apple Human Interface Guidelines : Windows
<http://developer.apple.com/Mac/library/documentation/UserExperience/Conceptual/AppleHIGuidelines/-XHIGWindows/XHIGWindows.html>
- SOFL** How to write OS X Finder plugin ?
<http://stackoverflow.com/questions/1294335/how-to-write-os-x-finder-plugin>
- LABL** Seeking Finder Plugin
<http://forums.macosxhints.com/showthread.php?t=57823>
- ASPL** Working with Spotlight
<http://developer.apple.com/macosx/spotlight.html>
- CMPX** Contextual Menu Plugins for Mac OS X
http://www5.wind.ne.jp/miko/mac_soft/contextual_menu_x/index-en.html
- SPRY** The life and death of Springy contextual menu plug-in
<http://www.springyarchiver.com/blog/topic/149>
- ACCS** UIElementInspector
<http://developer.apple.com/mac/library/samplecode/UIElementInspector/>
- INPM** NSInputManager Class Reference
http://developer.apple.com/legacy/mac/library/documentation/Cocoa/Reference/ApplicationKit/Classes/NSInputManager_Class/Reference/Reference.html
- SIXC** Step into Xcode, F. Anderson, Addison Wesley, 2006
- AHCB** Programmation Cocoa sous Mac OS X 3è édition, A. Hilleglass, Pearson Education, 2008
- SOHF** Stack Overflow : Hidden features of Objective-C
<http://stackoverflow.com/questions/211616/hidden-features-of-objective-c>
- MAQA** Friday Q&A 2010-01-29 : Method Replacement for Fun and Profit, M. Ash
<http://www.mikeash.com/pyblog/friday-qa-2010-01-29-method-replacement-for-fun-and-profit.html>

- SORT** STOP!!! Objective-C Run-TIME, N. Archibald
<http://felinemenace.org/~nemo/slides/eusecwest-STOP-objc-runtime-nmo.pdf>
- TCRM** Theocacao : Replacing Objective-C Methods at Runtime, S. Stevenson
<http://theocacao.com/document.page/266>
- CLSI** Cocoa with Love : Supersequent implementation, M. Gallagher
<http://cocoawithlove.com/2008/03/supersequent-implementation.html>
- RC07** RailsConf 2007 : the Ruby on Rails Podcast
<http://podcast.rubyonrails.org/programs/1/episodes/railsconf-2007>
- REMX** Reverse Engineering Mac OS X : How to dump an Apple protected binary
<http://reverse.put.as/2009/06/30/how-to-dump-an-apple-protected-binary/>
- OINJ** oin? oin! Ouvrir un .nib compilé avec Interface Builder, J. Aceituno
<http://j.oin.name/ouvrir-un-nib-compile-avec-interface-builder>
- UPFC** Fun Script : Ultimate proof that the Finder is now a Cocoa application, P. Mougín
<http://pmougín.wordpress.com/2009/10/28/ultimate-proof-that-the-finder-is-now-a-cocoa-application/>
- CCRE** Cocoa Reverse Engineering, M. Solomon
<http://www.culater.net/wiki/moin.cgi/CocoaReverseEngineering>
- SPUA** cocoa-dev mailing list archive : Re : NSURL from a sidebar plist entry
<http://lists.apple.com/archives/cocoa-dev/2006/Mar/msg02042.html>
- CFSC** Cocoa for Scientists : Getting Closure with Objective-C
<http://www.macresearch.org/cocoa-scientists-part-xxvii-getting-closure-objective-c>