Proposition pour un modèle à grains extrêmement fins

David Fauthoux
directeur: Jean-Paul Bahsoun
fauthoux@irit.fr, bahsoun@irit.fr
IRIT

21 mars 2002

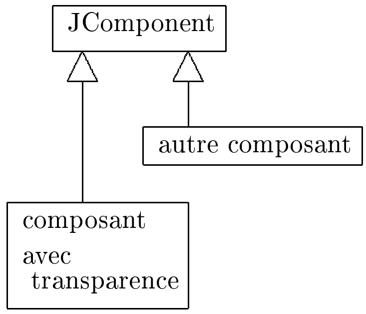
Résumé

Écrire une plate-forme orientée-objet demande de répondre à certains critères de réutilisabilité. Nous proposons ici un modèle à grains extrêmement fins qui donne au développeur un cadre riche et sûr. Ce modèle de "classe-lentille" localise et hiérarchise les propriétés de la plate-forme, et permet leur adaptation puis leur combinaison. En n'utilisant que des notions connues et en ne rajoutant pas de mot-clé au langage, le modèle peut être écrit et schématisé sous une forme rigoureuse mais intuitive.

1 Introduction

Actuellement, écrire un programme en langage orienté objet commence par l'utilisation de composants déjà écrits.
Par exemple, manipuler des fichiers en Java passe par l'utilisation du package `java.io`, les composants du package se chargeant des appels au système d'exploitation.
Un programme se construit donc à partir d'une (ou plusieurs) plate-forme. Une plate-forme regroupe un ensemble de composants destinés à être réutilisés par l'auteur du programme et combinés entre eux.

Considérons la plate-forme Swing [Mic02], utilisée pour la construction des interfaces graphiques, comme un exemple de programmation par composants.
Son utilisation est aisée et directe car chacun de ses composants (`JButton`, `JLabel`, ...) est assez riche pour convenir à une interface graphique classique. Mais cette plate-forme, à cause de sa structure, restreint fortement les possibilités du programmeur. Tous les composants de Swing héritent d'une classe racine (`JComponent`) qui possède toutes les propriétés communes et la gestion de l'interaction avec le système Swing (affichage, agencement des composants entre eux...). Supposons par exemple que nous créons un composant capable de s'afficher en transparence. Le code pour l'affichage est interne à la classe et inaccessible, la méthode `paint` des composants ne déléguant pas le travail d'affichage à une autre classe. Nous sommes donc forcés d'hériter la classe afin de réécrire le code de la méthode d'affichage. Dès lors, il n'est plus possible de porter cette propriété de transparence à d'autres composants puisqu'ils n'héritent pas de notre classe mais de `JComponent`.



Identifions dans Swing un autre problème qui restreint sa réutilisabilité. Certains composants possèdent des propriétés de taille (redimensionnement) mais pas tous. Certains autres des propriétés de couleur. Un `JButton` peut posséder un texte et une icône ; un `JMenuItem` également car ils héritent de la même classe, qui, elle, gère cette propriété. Les propriétés des composants sont donc déterminées par un système d'héritage. Il est donc impossible à un programmeur utilisant cette plate-forme de rattacher ou de détacher les propriétés qu'il désire, selon le programme à écrire. Il est par exemple impossible de faire tourner un bouton selon un axe (propriété de rotation), ou de refuser qu'un bouton puisse être redimensionné.

Une plate-forme doit selon nous atteindre certains objectifs de réutilisabilité que nous décrivons dans la section suivante. Dans les sections 3, 4 et 5, en nous basant sur le concept de Mixin[ALZ00] et en partant à l'opposé de l'écriture des classes-librairies, nous introduisons la notion de composant-lentille ou "lens". Dans les sections 6 et 7, nous écrivons, à partir de notre exemple principal, les classes-lentilles en Java et nous leurs attribuons un schéma explicite et clair. Nous enrichirons notre modèle vers les arbres dans notre conclusion.

2 Compétences que doit avoir une plate-forme réutilisable

Nous pensons que les objectifs à la construction d'une plate-forme réutilisable peuvent se diviser selon deux axes :
- Extensibilité : Les composants doivent être écrits en vue d'être adaptés, étendus.
- Composition : Les composants doivent être écrits en vue d'être composés [NM95].

Les exigences contenues dans la notion d'extensibilité sont l'adaptation aisée (pas de protocoles de programmation ni d'excès de rigueur de la part du programmeur), l'adaptation riche (laisser un maximum de liberté au programmeur pour adapter le composant), l'adaptation sûre (interdire au programmeur le plus tôt possible, si possible à la compilation, les utilisations faussées des composants), la maintenance ciblée (la correction ou l'amélioration d'un composant ne doit pas influencer les autres composants, ni ne doit conduire à la modification forcée d'autres composants).

Les exigences contenues dans la notion de composition sont la composition sûre (interdire au programmeur de composer des composants qui ne sont pas écrits pour, voir [NM95]), la composition maximale (permettre au programmeur de combiner tous composants ne l'interdisant pas), la composition tardive [BW, KAZ98] (partitionner au maximum les composants de la plate-forme, pour laisser le programmeur libre de composer seulement les propriétés voulues, et toutes les propriétés voulues).

Afin de montrer que l'ensemble de ces objectifs peut être atteint, nous utiliserons une notion clé, sésame de notre modèle, qui nous permettra de façonner les composants d'une plate-forme : l'interface.

La définition avant tout technique de l'interface (séparer l'implémentation de la déclaration) apporte aussi certaines notions conceptuelles qui montrent à quel point l'interface sera le ciment de notre travail :
- l'interface permettra de déclarer des variables sans les lier à une implémentation. Cette notion de généricité dynamique nous permettra d'exprimer qu'un composant peut se brancher sur un ensemble d'autres.
- Nous n'utiliserons pas le downcasting (retypage), ce qui nous permettra d'assurer que lorsqu'un variable est déclarée, seules les méthodes de son interface pourront être appelées, même si l'implémentation en possède des supplémentaires. Cette contrainte de typage fort nous assurera la sûreté de la composition : seul un composant ayant été prévu pour être combiné pourra l'être.
- L'utilisation de l'interface au lieu de la notion de classe nous permettra également de ne permettre l'accès qu'à certaines méthodes. L'interface agira comme bouclier, interdisant l'extension non sûre, ne permettant au programmeur de n'étendre ou de ne modifier le comportement que des méthodes de l'interface.

Donc l'interface nous donnera un cadre alliant liberté (généricité dynamique - late binding) et restriction (typage fort, bouclier - opacité).

3 Le point du vue Mixin

(voir [BW], page 206)

Nous utiliserons dans cet article une vision intuitive et simplifiée des mixins. Un mixin est une classe qui peut se brancher sur une classe mère dont la spécification correspond [ALZ00]. Un mixin permet ainsi de réutiliser le code de la classe mère, et force à écrire la classe fille (mixin) en fonction uniquement de la spécification de la classe mère.

Le mixin peut donc être vu comme un greffon sur une interface, qui vient ajouter ou modifier des fonctionnalités de cette interface. Nous conservons donc l'idée qu'un mixin permet de réutiliser n'importe quelle classe correspondant à une déclaration (interface), et ceci en forçant le programmeur de ce mixin à ne travailler qu'avec une spécification.

On reste ainsi dans nos objectifs de composition maximale (branchement sur toute classe correspondant à l'interface) et d'extensibilité sûre (écriture du mixin selon une interface). Nous forçons ainsi les extensions à être relative à une spécification et non à une implémentation. L'implémentation de chacun des composants sera indépendante de celle des autres composants. Cette indépendance des implémentations nous fait gagner la maintenance ciblée.

4 Une classe n'est pas une librairie

(voir [Kni99] §2.1, [FF99] §2, [BW] page 210)

Notre travail est basé sur l'idée suivante : ne pas voir les classes comme des librairies avec états ! Un point de vue général est de considérer un programme comme un regroupement de plusieurs parties, chaque partie remplissant un rôle, chaque partie étant modélisée par un objet. Chaque objet ainsi défini gère un ensemble de fonctionnalités, protège contre les erreurs possibles et contient souvent un sous-système d'exécution. Par conséquent, un tel objet est lourd (beaucoup de méthodes) et il contient un code complexe qui gère son comportement. Il est en fin de compte aussi lourd qu'une librairie, la seule différence étant cette gestion de comportement.

Nos observations :
- il est difficile de modifier ou d'adapter les composants lourds d'une plate-forme (voir par exemple

`JComponent` [Mic02]).
- La gestion interne et complexe des composants lourds rend difficile, voire impossible, leur maintenance.

Essayons d'identifier les obstacles qui limitent la réutilisabilité :
- avant d'adapter un composant à ses objectifs, le programmeur doit connaître toutes ses propriétés et son comportement.
- Référencer un objet depuis un autre n'exprime pas un accès partiel à cet objet. Le programmeur ne peut pas exprimer qu'il utilise dans certains cas qu'un sous-ensemble des méthodes de l'objet.
- Un objet lourd contient beaucoup de propriétés. Le programmeur ne peut pas choisir de n'utiliser qu'une seule de ces propriétés [WS]. En effet, le fonctionnement interne de l'objet provoque des dépendances entre ses propriétés. Par exemple, la capacité d'un bouton à être cliquable est en général rattachée à sa propriété d'affichage, s'il est recouvert, il n'est plus cliquable.
- L'adaptation d'un composant ne peut être anticipée lors de son écriture. Les développeurs font confiance aux techniques orienté-objet actuelles, au risque de casser le fonctionnement interne d'un objet en lui créant une classe qui en héritent et qui modifie ce comportement [MS98, WS].

Si nous voulons éviter ces écueils, nous ne pouvons donc accepter les composants lourds lors de l'écriture d'une plate-forme. C'est pourquoi nous proposons un modèle qui va à l'extrême opposé de l'option "classe-librairie" :
- nous n'autoriserons qu'une seule propriété par classe
- à l'inverse du point de vue de [FKF98], ajouter une propriété ne sera pas ajouter une méthode.

5 La notion de classe-lentille

Nous proposons ici un modèle à objets extrêmement fins. A chaque propriété de la plate-forme, nous associons une interface, ou "interface légère", qui ne déclare qu'une seule propriété. Toutes les classes implémentant cette interface mère assurent répondre à sa spécification. Une telle classe est un raffinement sémantique de l'interface [WS, NM95]. Par conséquent, nous pouvons considérer que l'interface mère déclare, en plus de la propriété globale, un ensemble de classes qui implémentent cette propriété d'une façon spécifique. Cet ensemble de petits composants assure que chacun de ces derniers peut être combiné avec les autres.

Ce modèle à grains fins permet au programmeur de combiner tous les raffinements de la propriété entre eux. "A wrapper can be wrapped again" (voir [BW] §5). Il peut ainsi définir de façon libre, mais sûre, une spécialisation de la propriété pour ses objectifs. Modifier ou étendre la propriété revient à agréger le micro-composant contenant cette modification.

Exemple

Soit une interface mère Drawer qui définit la propriété "capable d'être dessiné". Cette interface déclare une méthode d'affichage comme `draw` par exemple.

Soit la classe `ImageDrawer` qui implémente cette interface et donc spécialise la capacité d'être dessiné : cette classe dessine une maison.

Soit la classe `RotateDrawer` qui applique une rotation de 45° au dessin.

Soit enfin la classe `AlphaDrawer` qui dessine en semi-transparence.

Les deux dernières classes s'appliquent sur un composant Drawer. Elles définissent en effet une propriété de dessin relative à une autre. Leur façon de dessiner est de modifier un dessin.

Si nous voulons dessiner une maison penchée de 45°, il suffit de combiner un `RotateDrawer` sur un `ImageDrawer`, et d'appeler la méthode `draw`. Si nous voulons une maison semi-transparente, il faut utiliser un `AlphaDrawer` sur l'`ImageDrawer`. Si nous voulons une maison à la fois penchée et transparente, nous combinons les trois classes.

Nous pouvons donc décider des propriétés à appliquer qu'au moment voulu : composition maximale et tardive. Nous ne pouvons appliquer la rotation *que* sur un composant de type Drawer et nous *savons* que la rotation ne peut être appliqué que sur un tel composant : composition sûre et guidée. Si nous décidons d'améliorer l'algorithme de rotation, nous modifierons uniquement la classe `RotateDrawer` : adaptation aisée et maintenance ciblée. Nous pouvons créer un nouveau raffinement à la propriété. Par exemple, un micro-composant supplémentaire permettra de zoomer le dessin. Sans modifier aucune des classes existantes, nous pouvons ajouter la classe `ScaleDrawer` qui sera combinable avec tous les autres composants : adaptation riche et aisée.

6 Écrire notre modèle en Java

Nous proposons dans ce paragraphe une implémentation de notre modèle à grains extrêmement fins.

Nous avons choisi le langage Java parce qu'il inclut directement la notion d'interface comme nous l'entendons. Java est simple à lire et à écrire et enfin, Java correspond à notre philosophie : préférer restreindre le programmeur pour lui éviter les erreurs mais fournir un outil simple à utiliser. Comme dans Java, nous considérons que l'expressivité suit la simplicité. Notre défi consiste à implémenter notre modèle sans rajout de mot-clé au langage, il doit être directement et facilement utilisable. Deux rôles pour l'interface : montrer à l'extérieur la méthode accessible (et définir ainsi la propriété globale) et exprimer le branchement sûr lorsqu'une classe aura à se brancher sur un autre micro-

composant.
Écrivons notre exemple :

```
public interface Drawer {
    public void draw(Graphics g);
}

public class ImageDrawer implements Drawer {
    public ImageDrawer() {
    }
    public void draw(Graphics g) {
        // draw a house on g
    }
}

public class RotateDrawer implements Drawer {
    private Drawer wrappee;
    public RotateDrawer(Drawer wrappee) {
        this.wrappee = wrappee;
    }
    public void draw(Graphics g) {
        // rotate g of 45 degrees
        wrappee.draw(g);
    }
}
```

Cet exemple est construit à partir du patron suivant :
Pour définir l'interface mère :

```
public interface MotherInterfaceName {
    // method that declares the property:
    public ...
}
```

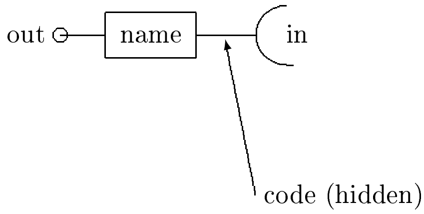
Et pour définir un raffinement de la propriété, défini relativement à celle-ci, nous écrivons un composant se branchant sur un autre :

```
public class RefinementName
  implements MotherInterfaceName {

    private MotherInterfaceName wrappee;

    public RefinementName(
       MotherInterfaceName wrappee) {
           this.wrappee = wrappee;
    }

    // method declared in interface:
    public ... {
           // algorithm to apply
           // refinement on wrappee
    }
}
```

Remarquons que notre implementation peut être rapprochée de celle de [YM01]. Notons également que notre modèle reprend la notion de wrapping défini dans les Generic Wrappers [BW], et que notre implémentation rejoint celle de Java pour les flux de données. Dans `java.io`, un `InputStream` définit la propriété (flux de données en lecture) et d'autres classes définissent les raffinements. Par exemple, `BufferedInputStream` mets les données en tampon avant l'envoi. `BufferedInputStream` se branche sur un `InputStream`.

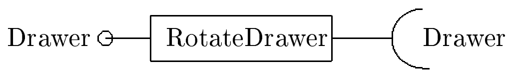
7 Schema

Afin d'exprimer clairement et d'un seul coup d'œil la composition des classes-lentilles, nous proposons le schéma suivant :

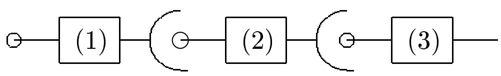


Branchement en out : l'interface implémentée par la lentille, ce qu'elle rend publique.
Branchement en in : sur quel type de la lentille celle-ci doit être branchée.
Ce schéma s'inspire du schéma pour les composants par Oscar Nierstrasz [NM95] avec les "in" et "out plugs". Il s'inspire également du "pictorial view of mixins" par Davide Ancona and Elena Zucca [AZ98].

Exemple :



Écrivons notre exemple en combinant les lentilles :



(1) : `RotateDrawer`
(2) : `AlphaDrawer`
(3) : `ImageDrawer`

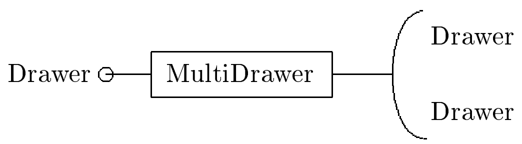
L'`ImageDrawer` possède un schéma simplifié puisqu'il n'a pas de branchement en in.

Conclusion

Nous avons essayé d'identifier les problèmes de réutilisabilité qui peuvent survenir lors de l'écriture d'une plate-forme. A partir de la notion de mixin, et à l'opposé des composants lourds, nous proposons un modèle à grain extrêmement fins qui donnent libre cours au programmeur de combiner toutes les propriétés proposées dans une plate-forme. Ce modèle de "classe-lentilles" répond aux exigences de réutilisabilité que nous avons posé comme point de départ.

Nous avons choisi ce terme de "lentille" car, lorsque les propriétés sont combinées entre elles et que leur schéma forme une ligne de raffinements, nous pouvons imaginer placer un œil devant la chaîne et voir à travers : les lentilles placées l'une sur l'autre ne modifie pas la propriété mais la raffine, comme une lentille réelle ne modifie pas la propriété de voir, mais peut changer le focus, la précision ou la distance de la vue. Enfin, comme une lentille réelle, une "classe-lentille" n'expose pas son fonctionnement interne : il n'est possible que de voir les *effets* de cette lentille. Et bien sûr, ce terme exprime en premier lieu la capacité d'une lentille à être combinée avec une autre pour agréger les propriétés entre elles.

Le modèle proposé ici est, pensons-nous, suffisamment restrictif pour éviter aux programmeurs la majorité des fautes qui casse la réutilisabilité, mais nous pouvons montrer qu'il n'est pas fermé et qu'il donne une puissance énorme à la notion de composition : en effet, nous pouvons désactiver la règle "un wrapper/un wrappee" des Generic Wrappers et autoriser une lentille à se brancher sur plusieurs autres lentilles. La structure d'aggrégation des propriétés peut donc devenir plus riche que la simple linéarité (qui, selon Wolfgang Weck [WS], n'est pas suffisante) et les lentilles peuvent être combinées en arbre. Par exemple, avec un composant qui affiche deux dessins l'un sur l'autre :



Actuellement, nous travaillons sur l'identification, grâce à notre modèle, de l'architecture d'un programme. Nous voulons montrer qu'un programme peut se structurer par un ensemble de flux, chaque flux étant une combinaison de micro-composants. Enfin, un de nos travaux futurs est d'approfondir la formalisation du modèle.

Références

[ALZ00] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam - a smooth extension of java with mixins. In *ECOOP*, pages 154–178, 2000.

[AZ98] Davide Ancona and Elena Zucca. A theory of mixin modules : Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4) :401–446, 1998.

[BW] Martin Büchi and Wolfgang Weck. Generic wrappers.

[FF99] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 94–104, 1999.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL 98 : The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.

[KAZ98] B. Kucuk, M. Alpdemir, and R. Zobel. Customizable adapters for blackbox components, 1998.

[Kni99] Günter Kniesel. Type-safe delegation for run-time component adaptation. In Rachid Guerraoui, editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628, pages 351–366. Springer-Verlag, New York, NY, 1999.

[Mic02] Sun Microsystems. java.sun.com, 2002.

[MS98] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In Eric Jul, editor, *ECOOP '98 — Object-Oriented Programming, 12th European Conference , Brussels, Proceedings*, volume 1445, pages 355–382. Springer-Verlag, 1998.

[NM95] Oscar Nierstrasz and Theo Dirk Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2) :262–264, 1995.

[WS] Wolfgang Weck and Clemens Szyperski. Do we need inheritance ?

[YM01] Sheik Yussuff and Permanand Mohan. An elegant implementation of business rules using functors in java. In *Proceedings of the 3rd International Conference on Enterprise Information Systems*, 2001.