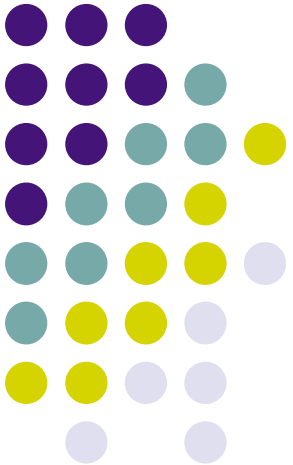
FSAB1402: Informatique 2
Techniques de Programmation Orientée Objet

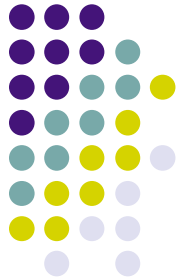
Peter Van Roy

Département d'Ingénierie Informatique, UCL

pvr@info.ucl.ac.be

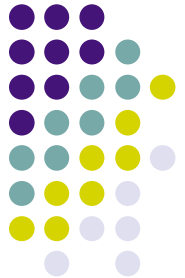


Ce qu'on va voir aujourd'hui



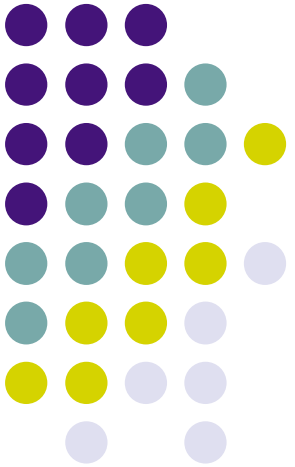
- Quelques techniques importantes de programmation orientée objet
- L'utilisation de l'héritage multiple
  - C'est aussi un exemple de polymorphisme
- Les diagrammes de classe UML
  - Un outil pour voir la structure statique d'un programme orientée objet en un coup d'oeil
- La programmation à grande échelle
  - Comment faire un programme correct

Suggestions de lecture pour ce cours

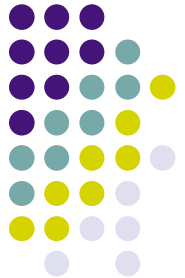


- Chapitre 6 (section 6.4):
  - Un exemple d'héritage multiple
  - Les diagrammes de classe UML
- Chapitre 5 (section 5.6)
  - La programmation à grande échelle

Résumé du dernier cours



Les objets et les classes

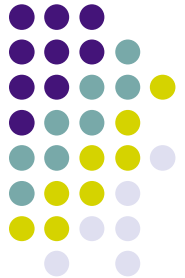


- Un objet est une collection de procédures ("méthodes") qui ont accès à un état commun
  - L'état est accessible uniquement par les méthodes
- L'envoi procédural: il y a un seul point d'entrée à l'objet, qui se comporte comme une procédure avec un argument (le "message")
- Les classes: cela permet de créer plusieurs objets avec les mêmes méthodes mais un autre état
- Une syntaxe pour les classes: cela facilite la programmation et garantit qu'il n'y a pas d'erreurs de forme dans la définition des classes

Le polymorphisme

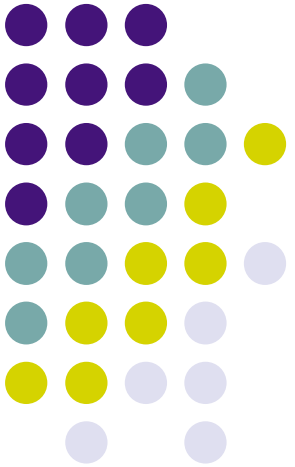
- Le polymorphisme est le concept le plus important (après l'abstraction!) dans la programmation orientée objet
- Des objets peuvent avoir la même interface mais une implémentation différente
  - {Line draw}, {Circle draw}, {Square draw}, …
- La même méthode peut marcher avec tous ces objets
  - Polymorphisme: la méthode accepte un argument de types (ici, de classes) différents. {F draw} peut marcher quand F est une ligne, un cercle, un carré, etc.
- Le principe de la répartition des responsabilités: chaque responsabilité est concentrée dans une partie du programme au lieu d'être morcelée partout
- Si chaque objet satisfait aux mêmes propriétés, cela marche!

L’héritage

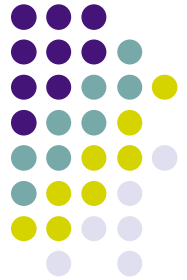


- La définition incrémentale des classes
  - Une classe est définie en prenant une autre comme base, avec des modifications et des extensions
  - Lien dynamique (le bon défaut) et lien statique (pour redéfinition)
- L’héritage est dangereux
  - La possibilité d’étendre une classe avec l’héritage est une autre interface à cette classe, une interface qui a besoin de maintenance comme les autres!
  - L’héritage versus la composition: nous recommandons d’utiliser la composition quand c’est possible
- Le principe de substitution
  - Si A hérite de B, alors toute procédure qui marche avec O_B doit marcher avec O_A
  - Avec ce principe, les dangers sont minimisés

L’utilisation de l’héritage multiple

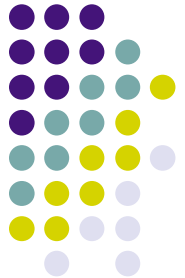


Un exemple de l'héritage multiple



- L'héritage multiple est important quand un objet doit être deux choses dans un programme
- Nous allons élaborer un exemple d'une librairie pour manipuler des figures graphiques
  - Lignes, cercles et figures plus complexes
- Nous allons définir une opération pour créer une figure qui est une collection de figures plus simples
  - Nous utiliserons l'héritage multiple pour définir cette opération
- L'idée de cet exemple vient de Bertrand Meyer dans son livre "Object-Oriented Software Construction". Ce livre montre bien l'utilité de l'héritage multiple.

Figures géometriques



- Nous définissons d'abord la classe Figure pour modéliser des figures géometriques dans un programme:

```
class Figure
   …
end
```

- Nous supposons qu'il y a trois méthodes pour chaque figure: init, move(X Y) et display
- Toutes les figures hériteront de Figure
  - Figure doit alors contenir les méthodes qui marchent pour toute figure

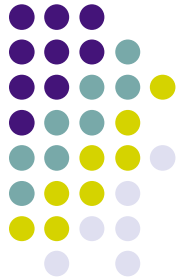
Définition de la classe Figure

- La classe Figure dans notre exemple sera très simple: elle donnera une erreur quand on essaie d'invoquer un objet avec une méthode nonexistante:

```
class Figure
      meth otherwise(M)
         {Browse 'Error: message '#M#' not understood'}
      end
end
```

- Dans le livre du cours, l'erreur est indiquée avec une exception (l'instruction **raise**)
  - Nous ne verrons pas les exceptions dans ce cours

Définition de la classe Line



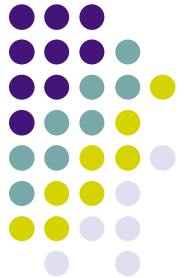
- La classe Line modélise les lignes droites:

```
class Line from Figure
      attr canvas x1 y1 x2 y2
      meth init(Can X1 Y1 X2 Y2)
         canvas:=Can
         x1:=X1 y1:=Y1
         x2:=X2 y2:=Y2
      end
      meth move(X Y)
         x1:=@x1+X y1:=@y1+Y
         x2:=@x2+X y2:=@y2+Y
      end
      meth display
         {@canvas create(line @x1 @y1 @x2 @y2)}
      end
end
```

Surface de dessin: canvas

- La définition de Line introduit le concept de “canvas”, qui est une surface de dessin
- Le canvas est défini par une librairie graphique du système Mozart, le module QTk
- Pour compléter l’exemple, nous allons utiliser QTk et le canvas
  - Nous ne verrons qu’une petite partie de ce que peut faire QTk. Pour plus d’informations, vous pouvez regarder chapitre 10 dans le livre du cours.

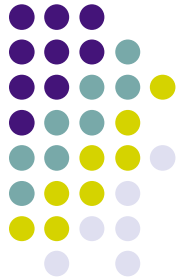
Définition de la classe Circle



- La classe Circle modélise les cercles:

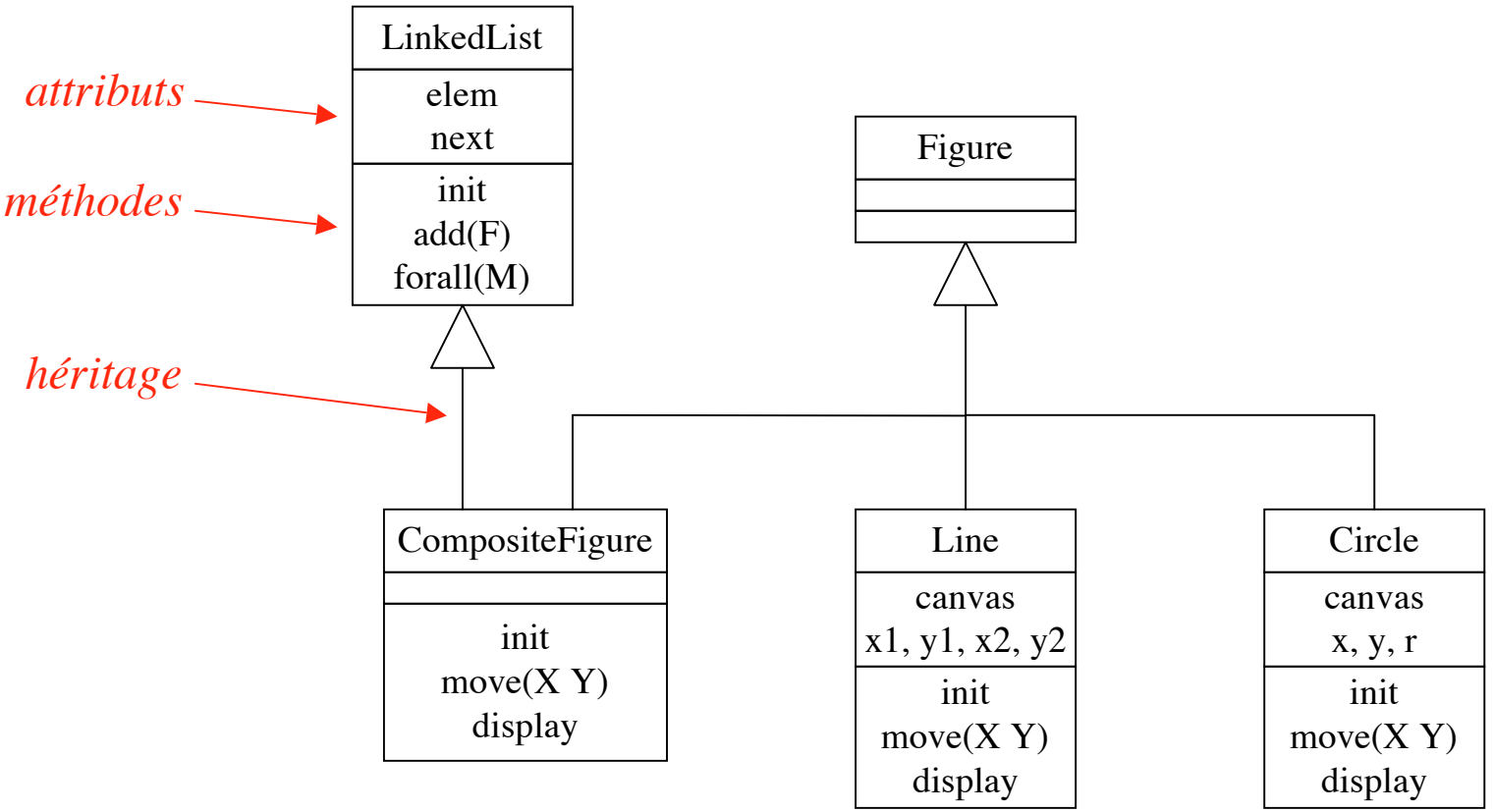
```
class Circle from Figure
      attr canvas x y r
      meth init(Can X Y R)
         canvas:=Can
         x:=X y:=Y r:=R
      end
      meth move(X Y)
         x:=@x+X y:=@y+Y
      end
      meth display
         {@canvas create(oval @x-@r @y-@r @x+@r @y+@r}
      end
end
```

Définition des figures composées



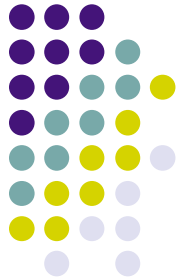
- Pour définir les figures composées, nous allons définir une classe CompositeFigure qui hérite de Figure et d'une autre classe, LinkedList
  - LinkedList définit une abstraction de liste enchainée
- Un objet de CompositeFigure est donc à la fois une Figure et une LinkedList!
  - Attention: l'héritage multiple ne marche pas avec toutes les classes. L'héritage multiple marche ici parce que Figure et LinkedList sont complètement indépendantes.
- Maintenant que nous avons toutes les classes, nous pouvons faire un résumé de la structure de notre librairie, dans un diagramme de classe
  - Le diagramme de classe fait partie d'une méthodologie de construction de programmes orienté-objet, appelé UML (Uniform Modeling Language)

Diagramme de classe



- On peut décrire la structure de la librairie avec un diagramme, appelé “diagramme de classe”

Définition de la classe LinkedList



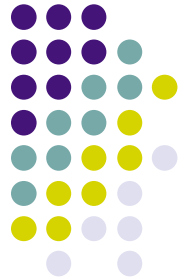
- La classe LinkedList implémente une liste enchainée:

```
class LinkedList
      attr elem next
      meth init(elem:E<=null next:N<=null)
         elem:=E next:=N
      end
      meth add(E)
         next:={New LinkedList init(elem:E next:@next)}
      end
      meth forall(M)
         if @elem\=null then {@elem M} end
         if @next\=null then {@next forall(M)} end
      end
end
```

Commentaires sur la définition de LinkedList

- Il y a trois méthodes: init, add(E) et forall(M)
- La méthode init utilise des initialisations à option
  - Si on appelle init sans mentionner elem, la valeur null sera donnée par défaut (et de même pour next)
- La méthode add(E) ajoute E au début de la liste
- La méthode forall(M) invoque chaque objet de la liste avec le message M

Définition de la classe CompositeFigure



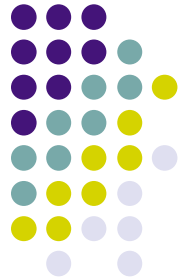
- La classe CompositeFigure modélise une figure qui est fait d'une collection d'autres figures:

```
class CompositeFigure from Figure LinkedList
      meth init
         LinkedList,init
      end
      meth move(X Y)
         {self forall(move(X Y))}
      end
      meth display
         {self forall(display)}
      end
end
```

Commentaires sur la définition de CompositeFigure

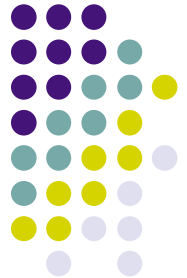
- La classe CompositeFigure a trois méthodes, init, move(X Y) et display, comme toute figure
- La méthode init est une redéfinition
  - L'utilisation d'un lien statique est nécessaire pour initialiser LinkedList
- Les définitions de move(X Y) et display sont un bel exemple de polymorphisme
  - Elles utilisent la méthode forall de LinkedList. La généralité de forall se montre très utile!

La beauté de l'héritage et du polymorphisme



- Est-ce que vous appercevez la beauté de cette définition de CompositeFigure?
- Une figure peut être une collection d'autres figures, et certaines de ces figures peuvent elles-mêmes être des collections d'autres, et ainsi de suite
- La structure de l'héritage et l'utilisation du polymorphisme garantissent que tout marchera toujours bien
  - Toutes les classes, Line, Circle et CompositeFigure, comprennent les messages move(X Y) et display

Exécution de l'exemple (1)



- Nous définissons d'abord un canvas avec le module QTk (*) :

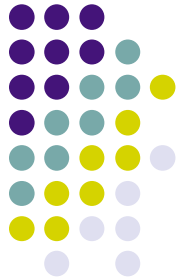
```
declare
W=250 H=150 Can
Wind={QTk build
                td(title: "Simple graphics"
                   canvas(width:W height:H bg:white handle:Can)}
{Wind show}
```

- Ceci définit une fenêtre qui contient un canvas avec largeur 250 pixels et hauteur 150 pixels

- (*) Pour utiliser QTk il faut d'abord le charger:

```
declare [QTk]={Module.link ["x-oz://system/wp/QTk.ozf"]}
```

Exécution de l'exemple (2)



- Maintenant nous pouvons définir une figure composée:

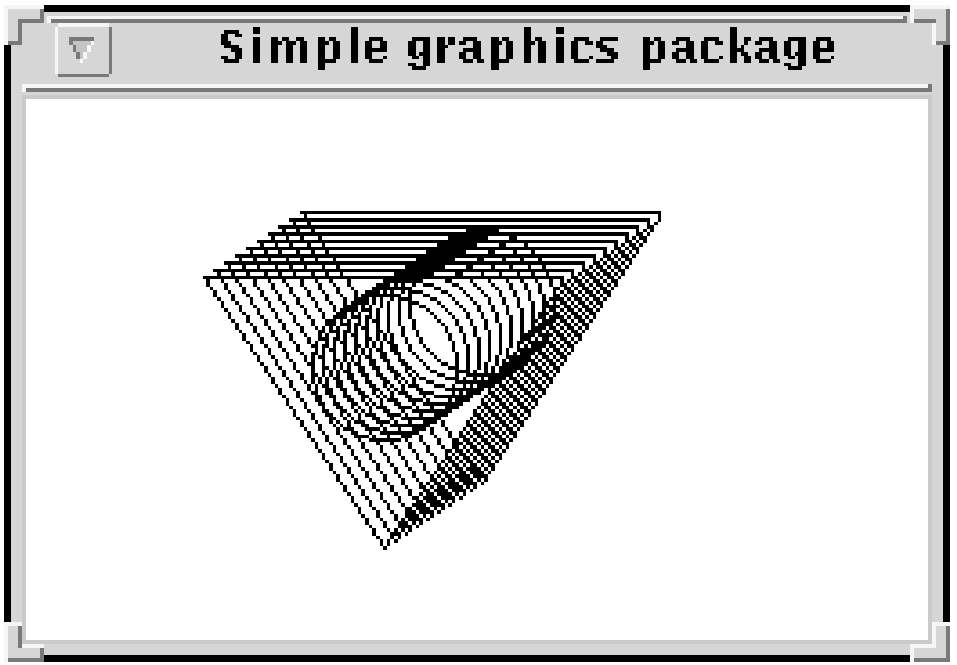
```
declare
F1={New CompositeFigure init}
{F1 add({New Line init(Can 50 50 150 50)})
{F1 add({New Line init(Can 150 50 100 125)})
{F1 add({New Line init(Can 100 125 50 50)})
{F1 add({New Circle init(Can 100 75 20)})
```

- Pour afficher cette figure: {F1 display}
- Pour déplacer cette figure: {F1 move(10 10)}

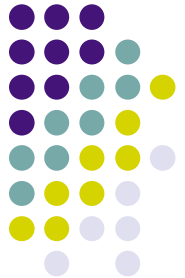
Exécution de l'exemple (3)

- Voici une série de commandes qui affiche un dessin plus compliqué:

```
for I in 1..10 do
      {F1 display}
      {F1 move(3 ~2)}
end
```



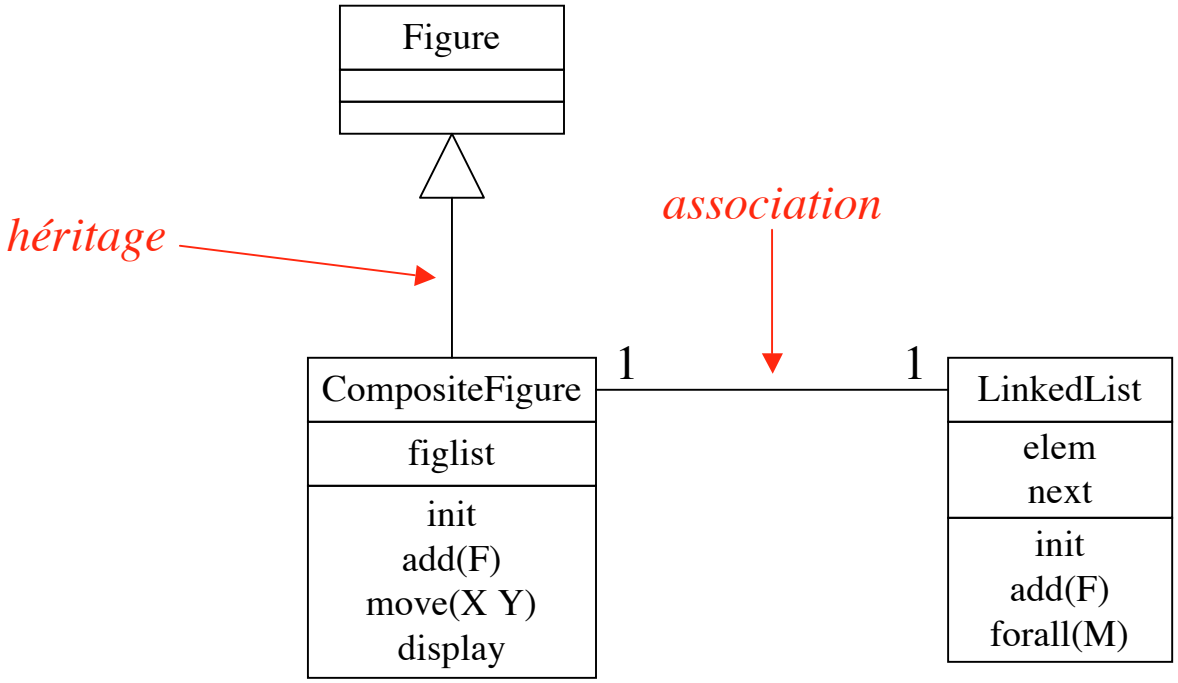
Une définition alternative



- Il est possible de définir CompositeFigure avec l'héritage simple:

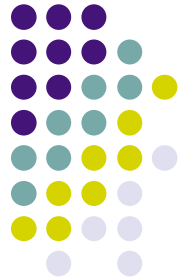
```
class CompositeFigure from Figure
      attr figlist
      meth init
         figlist:={New LinkedList init}
      end
      meth add(F)
         {@figlist add(F)}
      end
      meth move(X Y)
         {@figlist forall(move(X Y))}
      end
      meth display
         {@figlist forall(display)}
      end
end
```

Diagramme de classe pour la définition alternative



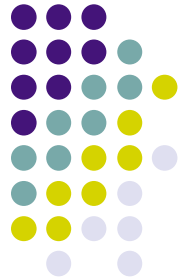
- Chaque CompositeFigure contient une LinkedList. Cette relation s'appelle une association.

Limites des diagrammes de classe



- Les diagrammes de classe sont omniprésents dans la programmation orienté-objet industrielle
  - Il faut faire attention à bien comprendre leurs limites
- Ils spécifient la structure statique d'un ensemble de classes
  - Ils ne spécifient pas le comportement d'une classe
    - Par exemple, la séquence d'invocations entre deux objets
  - Ils ne spécifient pas les invariants du programme (spécification)
    - Une spécification est une formule mathématique, pas un programme!
- Ils ne modélisent qu'un niveau dans la hiérarchie d'un programme (voir plus loin dans ce cours)
  - Un programme a généralement plusieurs niveaux

Commentaires sur la définition alternative

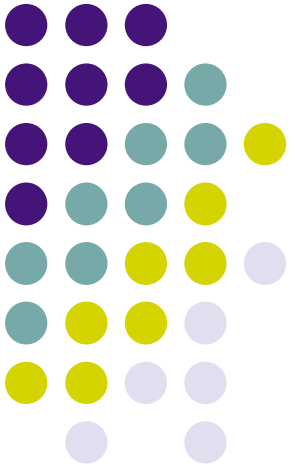


- Cette définition utilise la composition: il y a un attribut figlist qui contient un objet LinkedList
- Il y a une quatrième méthode, add(F), dans CompositeFigure, qui n'est pas dans la spécification de Figure
- Comparaison des deux définitions:
  - Héritage multiple: chaque CompositeFigure est aussi une LinkedList. On peut faire des calculs LinkedList directement sur les CompositeFigure.
  - Héritage simple: la structure de CompositeFigure est complètement cachée. Cela protège des CompositeFigure des calculs autres que des calculs pour les figures.

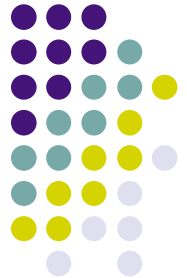
Mise à l'échelle de l'exemple

- La librairie qu'on a défini est très simple
- Qu'est-ce qu'il faut faire pour en faire une vraie librairie de grandeur réelle?
  - Un plus grand nombre de figures
  - Mettre le canvas ailleurs (un argument de display)
  - Création d'un "journal" (créer une figure comme une séquence de commandes)
  - Plus de souplesse dans le display
  - … ?

Programmation à grande échelle (“in the large”)

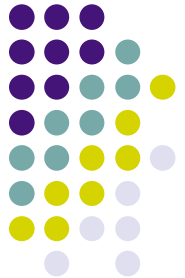


Programmation à grande échelle



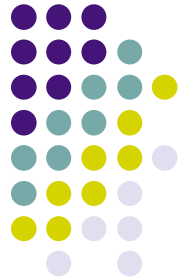
- Par définition, la programmation à grande échelle (“in the large”) est la programmation par une équipe
  - C’est ici que le génie logiciel (“software engineering”) devient très important
  - Par définition, la programmation à petite échelle (“in the small”) est la programmation par une seule personne
- Il s’agit d’avoir une discipline qui permet de gérer des programmes dont chacun ne connaît qu’une partie
  - La psychologie joue une grand rôle: garder une équipe motivée, loyale, heureuse, etc. Cela fait partie d’un bon management.
  - Mais il y a aussi quelques règles plus techniques…
- On va donner des règles qui marchent avec de petites équipes (jusqu’à une dizaine de personnes)
  - Pour de plus grandes équipes (milliers de personnes pour certains projets industriels comme Airbus, etc.) il y a encore d’autres règles!

Gérer l'équipe



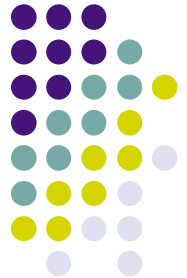
- La première chose est de s'assurer que l'équipe est bien coordonnée:
  - La responsabilité de chaque personne doit être compartimentalisée (par exemple, limitée à un composant ou un module)
  - Par contre, les connaissances doivent être échangées librement (pas d'informations secrètes!)
  - Les interfaces entre les composants doivent être soigneusement documentées. Une bonne documentation est un pilier de stabilité.

Méthodologie de développement



- On ne peut pas tout prévoir!
  - Une méthodologie où on essaie de tout spécifier dès le début est vouée à l'échec!
- Une méthodologie qui marche bien est le développement incrémental (aussi appelé développement itératif)
  - On fait pousser un programmer comme un être vivant: on commence petit et on grandit

Développement incrémental

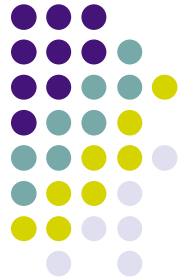


- On commence avec un petit nombre d'exigences. On fait alors une petite spécification de programme, et on fait un petit programme.
- Ensuite, on ajoute des exigences au fur et à mesure, selon l'expérience avec le programme précédent. A chaque itération, on fait un programme qui tourne et qui peut être testé et évalué par ses utilisateurs potentiels.
- Ne pas "optimiser" pendant le développement! Ne pas faire un programme plus compliqué uniquement pour augmenter sa vitesse. Utilisez des algorithmes simples avec une complexité acceptable.
  - L'optimisation prématurée est à l'origine de tout mal
  - On optimise à la fin, en mésurant les performances
- Pendant le processus, il faut parfois reorganiser le programme parce qu'on réalise qu'il est mal structuré. Cela s'appelle "refactoring".

Faire un programme correct: à la conception (raisonnement)

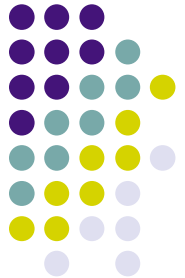
- Les exigences (“requirements”)
  - Il faut d’abord bien comprendre le problème qu’on veut résoudre!
- La spécification
  - Il faut bien spécifier le comportement du programme qui va résoudre notre problème
  - Souvenez-vous des trois piliers:
    - La spécification est une formule mathématique
    - Le programme est un code dans un langage de programmation
    - La sémantique fait le lien entre les deux
- Exemple de Account
  - Spécification: {A getBal(B)} {A transfer(S)} {A getBal(B’)} => B+S=B’ (un invariant)
  - Programme: implémentation de la classe Account
  - Preuve que le programme satisfait l’invariant: avec la sémantique

Conception par contrat ("design by contract")



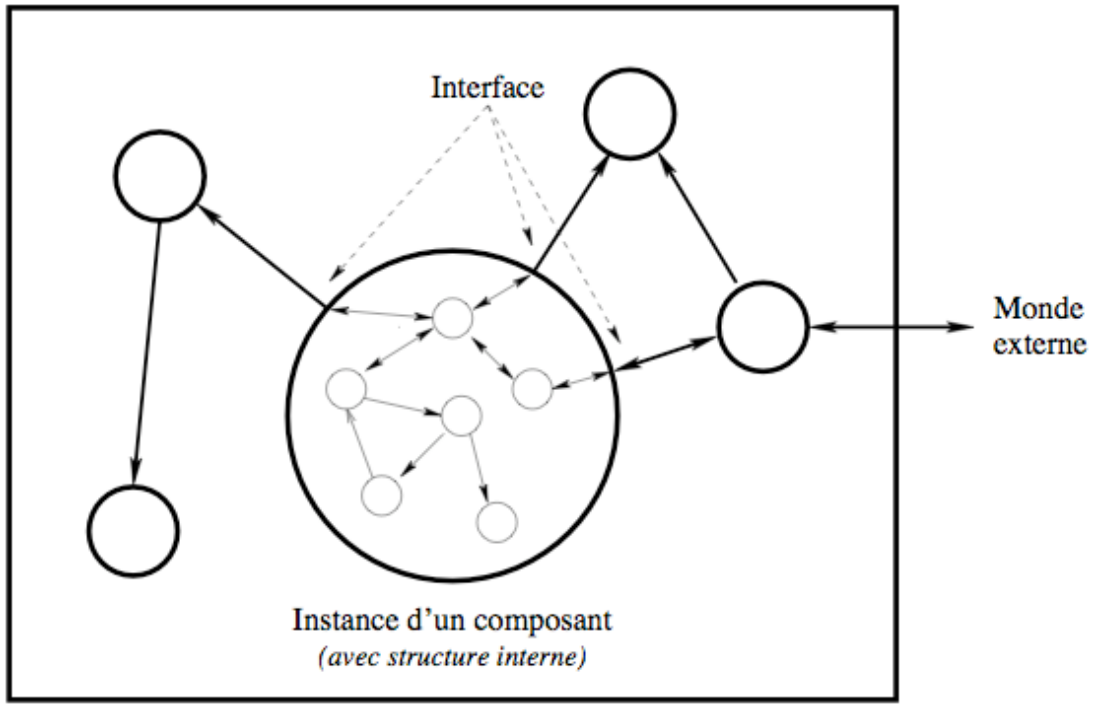
- Une méthode de concevoir de programmes corrects, développé principalement par Bertrand Meyer pour le langage Eiffel
- Idée: chaque abstraction de données implique un contrat entre le concepteur et les utilisateurs de l'abstraction
  - Les utilisateurs doivent garantir que l'abstraction est appelée correctement
  - En retour, le concepteur garantit que l'abstraction se comporte correctement
  - L'abstraction peut souvent vérifier si l'utilisateur l'appelle correctement. Ces vérifications sont faites à l'interface, pas dans l'intérieur qui alors n'est pas encombrée.
- Il y a une analogie avec la société humaine
  - Contrats entre personnes, contrôle à la frontière

Faire un programme correct: à l'exécution (tests)



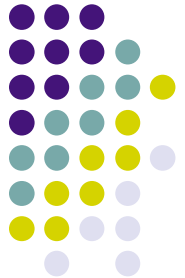
- En général, il est essentiel de faire des tests
  - Principe: Toute partie du programme qui n'a pas été testée ne marche pas
- Différentes formes de test
  - Unit test: tester chaque partie d'une application individuellement
    - C'est possible avec l'interface interactive de Mozart
  - Application test: tester l'application comme un tout
    - Par exemple, des tests d'utilisation par un utilisateur
- On peut dire beaucoup plus sur les tests

Structure du programme



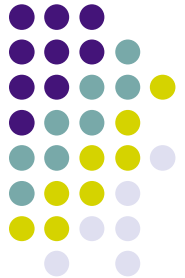
- En général, on utilise une structure hiérarchique
- A chaque niveau, l'application est un graphe de composants, avec des interfaces entre les composants
- Chaque composant est lui-même un graphe à l'intérieur

Les composants: quelques conseils



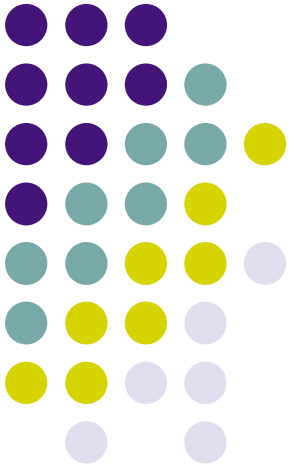
- Communication entre composants:
  - Il faut choisir le protocole de communication
  - Voir le transparent suivant
- Compilation efficace versus exécution efficace
  - Exécution efficace: les composants sont compilés ensemble (avec une analyse globale du compilateur)
  - Compilation efficace: chaque composant est compilé indépendamment (aucune analyse)
- Faciliter la maintenance
  - La maintenance se fait pendant toute la durée de vie du composant
  - De préférence, chaque décision de conception devrait être cachée dans un seul composant (par exemple, le format des données qui est connu uniquement par une seul composant). On dit que le composant a un "secret".
  - Il faut éviter de changer les interfaces

Communication entre composants

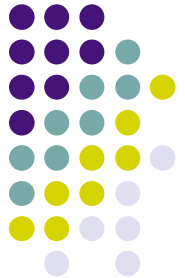


- Appel de procédure/fonction/objet: Le composant qui appelle l'autre s'arrête et attend une réponse (par exemple, un client/serveur)
- Coroutine: Quand il est appelé, chaque composant continue là où il s'était arrêté la dernière fois (par exemple, deux joueurs d'échecs)
- Concurrent et synchrone: Chaque composant exécute de façon indépendante des autres. Quand un composant appelle un autre, il attend le résultat. (deux joueurs d'échecs qui réfléchissent tout le temps)
- Concurrent et asynchrone: Chaque composant exécute de façon indépendante des autres. Un composant peut envoyer un message à un autre sans attendre le résultat. (une équipe de football)
- Boîte à courrier (variante de concurrent asynchrone): Un composant dépose un message dans la boîte d'un autre. L'autre peut lire uniquement les messages qui l'intéresse (choisis avec pattern matching).
- Modèle de coordination (variante de concurrent asynchrone): Un composant dépose un message dans un ensemble commun. Un autre le prend. Les deux composants ne se connaissent pas. (par exemple, un groupe d'imprimantes)

Résumé



Résumé



- L'héritage multiple et l'héritage simple
  - Définition de CompositeFigure qui est à la fois une Figure et une LinkedList
- Les diagrammes de classe UML
  - Ils permettent de voir en un coup d'oeil la structure statique du programme
  - Limites des diagrammes de classe
- La programmation à grande échelle ("in the large")
  - Organisation de l'équipe, le développement incrémental, la structure hiérarchique, les composants
  - Faire un programme correct:
    - Le raisonnement (a priori): conception par contrat
    - Les tests (a posteriori): unit test, application test