

## Introduction à la programmation objet

Jean-Philippe Domenger  
Université Bordeaux I & LaBRI

## Objectifs de ce cours

- Présenter les principaux concepts utilisés par les langages objets :
  - Modularité et encapsulation.
  - Objets, classes, et messages.
  - Interfaces, héritage, et polymorphisme.
  - Technique de réutilisabilité.
  - Approfondissement de l'héritage.
  - La réutilisation

JP Domenger, G.Eyrolles, F. Pellegrini 2

## L'évolution du génie logiciel

- Passage de la programmation à petite échelle vers la programmation à grande échelle
  - Evolution des langages de programmation
  - Evolution de la technologie
- Accroissement de la complexité
  - Implique la nécessité de l'abstraction

JP Domenger, G.Eyrolles, F. Pellegrini 3

## Les Langages au cours du temps

- 1ière Génération 1954-1958
  - Fortran 1    expressions mathématiques
  - Algol 58    expressions mathématiques
  - FLOWMATIC    expressions mathématiques
  - IPL V        expressions mathématiques

JP Domenger, G.Eyrolles, F. Pellegrini 4

## Les Langages au cours du temps

- 2ième Génération 1959-1961
  - Fortran II  
*sous programmes, compilation séparée*
  - Algol 60  
*structure en blocs, types de données*
  - COBOL  
*descriptions de données, gestion de fichiers*
  - LISP  
*traitement de listes, pointeurs, ramasse-miettes.*

JP Domenger, G.Eyrolles, F. Pellegrini 5

## Les Langages au cours du temps

- 3ième Génération 1962-1970
  - PL/I        *Fortran + COBOL + ALGOL*
  - Algol 68    *successeur rigoureux d'Algol 60*
  - Pascal      *successeur simple d'Algol 60*
  - C            *successeur de B*
  - Simula     *classes, abstraction de données*

Ancêtre des langages Objets

JP Domenger, G.Eyrolles, F. Pellegrini 6

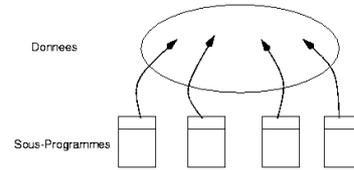
## Les Langages au cours du temps

- 4ième Génération 1970-
  - ADA *successeur de Pascal et Algol 68*
  - SmallTalk *successeur de Simula*
  - C++ *mariage Simula et C*
  - Objective C *variante à C++*
  - Eiffel *dérivé de Simula et ADA*
  - CLOS *LISP + Flavours*
  - Java *dérivé de Simula et C.*

JP Domenger, G.Eyrolles, F Pellegrini

7

## Topologie 1,2 Génération

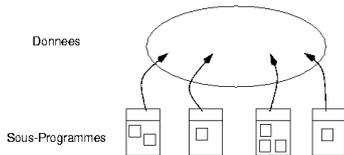


- Sous programmes = blocs élémentaires
- Pas de hiérarchie
- Propagations d'erreurs
- Extensibilité difficile

JP Domenger, G.Eyrolles, F Pellegrini

8

## Topologie 2,3 Génération

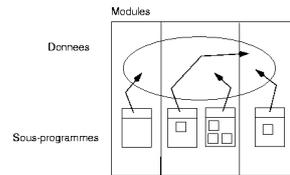


- Séparation des données globales
- Sous programmes = boîtes noires
  - Passages de paramètres
  - Retour de fonctions, imbrications de fonctions
- Découpage difficile du développement

JP Domenger, G.Eyrolles, F Pellegrini

9

## Topologie 3 Génération

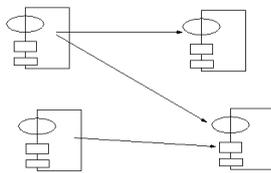


- Séparation des données locales, globales
- Abstraction modulaire
  - Interface Modulaires
  - Données internes
  - Fonctions internes

JP Domenger, G.Eyrolles, F Pellegrini

10

## Topologie Objets

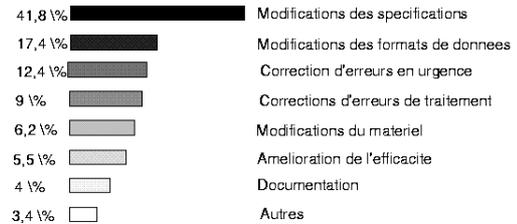


- Architecture basée sur les données (**Objets**)
- Types Abstrais (**Classes**)
- Protections de données (**Encapsulation**)
- Hiérarchie de classe (**Héritage**)

JP Domenger, G.Eyrolles, F Pellegrini

11

## Répartitions des coûts de maintenance



## Facteurs externes de qualité

- **Validité:** aptitude à réaliser les tâches définies par les spécifications
- **Robustesse:** aptitude à fonctionner dans des conditions non spécifiées
- **Extensibilité:** facilité d'adaptation aux changements de spécifications
- **Réutilisabilité:** aptitude à être réutilisé en tout ou partie pour de nouvelles applications

JP Domenger, G.Eyrolles, F Pellegrini

13

## Approche fonctionnelle versus approche objet

**On veut réaliser un éditeur de dessin. On crée des cercles, des rectangles, .....**

**On les déplace, on les agrandit, ....**

## Approche fonctionnelle

- Découpage en fonction des traitements.  
Un module = Un traitement
- Traitements
  - | Initialisation
  - | Déplacement
  - | Agrandissement
  - | .....

JP Domenger, G.Eyrolles, F Pellegrini

15

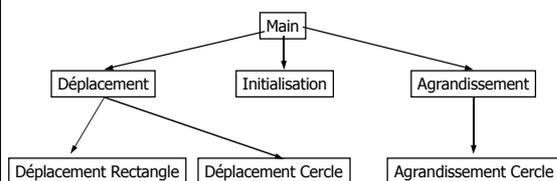
## Approche fonctionnelle

- Les données circulent à travers les traitements
  - | Données
    - | Liste des cercles
    - | Listes des rectangles

JP Domenger, G.Eyrolles, F Pellegrini

16

## Exemple de structuration fonctionnelle



JP Domenger, G.Eyrolles, F Pellegrini

17

## Exemple de pseudo code

```

Module Initialisation
Carre [] listeCarre;
Cercle [] listeCercle;
void initialisation() {
    ajouterCarre (listeCarre);
    ajouterCercle (listeCercle);
}

Module Déplacement
void Déplacement (listeCarre, listeCercle) {
    DéplacementCarre (listeCarre);
    DéplacementCercle (listeCercle);
}
    
```

JP Domenger, G.Eyrolles, F Pellegrini

18

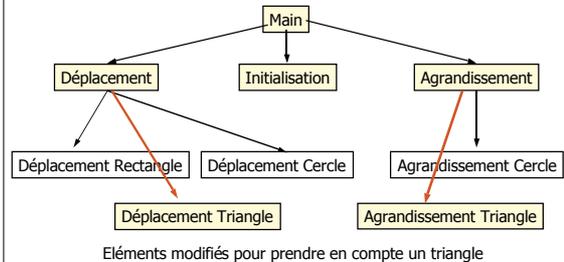
## Approche Fonctionnelle

- Avantages
  - Adapté à un algorithme
- Inconvénients
  - Programme difficilement exprimable par un unique algorithme
  - basée uniquement sur les traitements
  - partage d'une même structure de données
  - extensibilité et réutilisabilité sont pénalisées

JP Domenger, G.Eyrolles, F Pellegrini

19

## Extensibilité ?



JP Domenger, G.Eyrolles, F Pellegrini

20

## Approche Objet

- Regroupe dans le même module, le traitement et les données.
- Les objets représentent des abstractions du domaine du problème.
- Un programme objet consiste à faire collaborer des objets pour réaliser un traitement.

JP Domenger, G.Eyrolles, F Pellegrini

21

## Modularité

**La modularité est permise par l'utilisation de l'abstraction et de l'encapsulation.**

## Modularité

- Technique de décomposition visant à réduire la complexité d'un système en le considérant comme un assemblage de sous-systèmes plus simples
- Un module est un sous-système dont le couplage avec les autres est faible par rapport au couplage de ses propres parties
- La capacité à modulariser dépend du degré de couplage entre les sous-systèmes

JP Domenger, G.Eyrolles, F Pellegrini

23

## Abstraction

- Une abstraction fait ressortir les caractéristiques essentielles d'un objet
- Définition des frontières conceptuelles par rapport au point de vue de l'observateur.
- L'abstraction est représentée par les types abstraits.

JP Domenger, G.Eyrolles, F Pellegrini

24

## Type abstrait (1)

- Un type abstrait est composé :
  - D'un ensemble de valeurs possibles
  - **Le comportement**, un ensemble d'opérations applicables sur ce type.
    - Constructeurs
    - Accesseurs
    - Modificateurs d'état

JP Domenger, G.Eyrolles, F Pellegrini

25

## Type abstrait (2)

### ■ Exemple : type abstrait Point

- Construction
 

```
point: Float X, Float Y ⇨ Point
```
- Accesseurs
 

```
abscisse: Point ⇨ Float
ordonnée: Point ⇨ Float
```
- Modificateurs d'état
 

```
translater: Point P, Float DX, Float DY ⇨ Point
pivoter: Point P, Point C, Float A ⇨ Point
```
- Sémantique de composition
 

```
abscisse(translater (p, dx, dy))
           □
abscisse (p) + dx et ordonnée(p) + dy
```

JP Domenger, G.Eyrolles, F Pellegrini

26

## Encapsulation (1)

- Technique favorisant la modularité des sous-systèmes, par séparation de l'interface d'un module de son implémentation
- Interface (partie publique)
  - Liste des services offerts
  - Présentation du type Abstrait
- Implémentation (partie privée)
  - Réalisation des services offerts
    - Structures de données
    - Algorithmes et implémentation

JP Domenger, G.Eyrolles, F Pellegrini

27

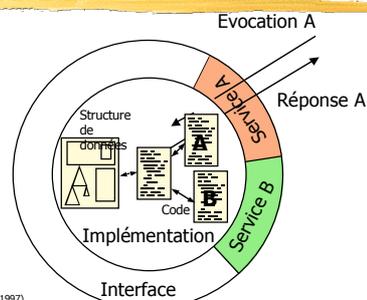
## Encapsulation (2)

- L'encapsulation doit être assurée :
  - Au niveau du langage : mots-clés qualificateurs **public, private, ...**
  - Au niveau de l'exécution : interdiction de manipulations hasardeuses de pointeurs, ...
- Les modules communiquent par évocation du comportement et non par accès mutuels à leurs données

JP Domenger, G.Eyrolles, F Pellegrini

28

## Module



D'après G. Falquet (1997)

JP Domenger, G.Eyrolles, F Pellegrini

29

## Intérêt de la Modularité

- Séparation entre les services et leur réalisation.
  - Evolution de l'implémentation du fournisseur sans remise en cause des clients.
- Compatibilité ascendante de l'interface
  - Services fournis doivent perdurer

JP Domenger, G.Eyrolles, F Pellegrini

30

## Intérêt de la Modularité

- Programmation contractuelle
  - Pré-Condition, vérification par le module que l'appel est conforme au contrat. Evite la propagation des erreurs.
  - Post-Condition, phase de mise au point  
Vérifie que l'objet est dans un état conforme.

JP Domenger, G.Eyrolles, F Pellegrini

31

## Objets et classes

**Comment les langages objets formalisent les notions de modules, d'encapsulation, et de typage**

## L'âge de raison

- L'approche orientée-objets a 30 ans
  - 1966 Une idée à Oslo (naissance de Simula)
  - 1969 La recherche à Xerox Park
  - 1980 Version industrielle de Smalltalk
  - 1986 1200 personnes pour OOPSLA'86
  - 1996 Omniprésence des solutions objets
  - 2006 Stabilisation de la transition vers l'objet

JP Domenger, G.Eyrolles, F Pellegrini

33

## Approche orientée objets

- Synthétise les notions de module et de type abstrait
- Permet la structuration du logiciel en termes de relations clients/fournisseurs
- Permet le découplage entre spécification et implémentation
- Polymorphisme simple
  - Héritage + liaison dynamique

JP Domenger, G.Eyrolles, F Pellegrini

34

## Concepts clés

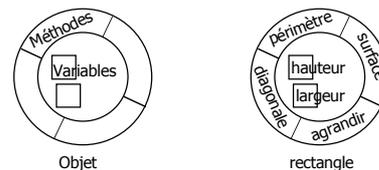
- Un programme est organisé comme un ensemble coopérant d'objets
- Chaque objet est instance d'une seule classe
- Une classe est la concrétisation d'une ou plusieurs interfaces
- Chaque interface représente un type
- Les types sont ordonnés partiellement par la relation d'héritage

JP Domenger, G.Eyrolles, F Pellegrini

35

## Objet (1)

- Entité contenant des données (état) et des procédures associées (comportement)



JP Domenger, G.Eyrolles, F Pellegrini

36

## Objet (2)

- **Objet** ⇔ instantiation de modules
  - Partie privée : valeur des variables (d'instance)
  - Partie publique : noms et prototypes des méthodes publiques
  - Communication : invocation (appel) de méthodes, retour de résultats

JP Domenger, G.Eyrolles, F Pellegrini

37

## L'identité

- Tout objet possède une identité qui lui est propre et qui le caractérise. L'identité est liée à la référence.
- L'identité permet de distinguer tout objet de façon non ambiguë, indépendamment de l'état.
- L'adresse mémoire n'identifie pas systématiquement l'objet.

JP Domenger, G.Eyrolles, F Pellegrini

38

## Etat

- Chaque objet a un état qui lui est propre. L'état regroupe les valeurs instantanées de tous les attributs d'un objet.
- L'état d'un objet peut évoluer au cours du temps
- L'état d'un objet représente les effets cumulés de son comportement

JP Domenger, G.Eyrolles, F Pellegrini

39

## Comportement

- Décrit les actions et les réactions d'un objet
- L'ensemble des opérations applicables à un objet définit son comportement
- Les opérations peuvent soit :
  - Accéder à l'état de l'objet sans le modifier
  - Modifier l'état de l'objet

JP Domenger, G.Eyrolles, F Pellegrini

40

## Etat et Comportement

- Le comportement d'un objet peut faire évoluer son état.
  - Pacman entre en collision avec une pastille.
- L'état d'un objet peut influencer le résultat produit par le comportement d'un objet.
  - Pacman entre en collision avec un fantôme
    - Si invulnérable ==> destruction fantôme
    - Si vulnérable ==> destruction pacman

JP Domenger, G.Eyrolles, F Pellegrini

41

## Encapsulation

- Un objet contient à la fois son état et son comportement
- Seules les opérations appartenant au comportement d'un objet peuvent modifier son état (*Encapsulation pure et dure*)
- Séparation entre interface et implémentation

JP Domenger, G.Eyrolles, F Pellegrini

42

### Message (1)

- Les objets collaborent en échangeant des messages.
  - P.translation(3,2) : envoyer à l'objet P le message « translation » avec les paramètres (2,3).
- Tout message reçu et accepté par un objet donné déclenche l'exécution d'une méthode

JP Domenger, G.Eyrolles, F.Pellegrini

43

### Message (2)

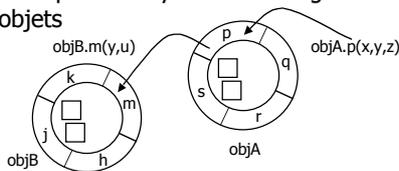
- La méthode à exécuter est choisie en fonction:
  - Du type du message (*nom de la méthode*)
  - Du type des paramètres contenus dans le message (*signature de la méthode*)
  - Du type réel de l'objet
    - *liaison dynamique*
  - Du type déclaré de l'objet
    - *liaison statique*

JP Domenger, G.Eyrolles, F.Pellegrini

44

### Message (3)

- Une méthode peut envoyer des messages à d'autres objets



- Un système est constitué d'objets communiquant entre eux

JP Domenger, G.Eyrolles, F.Pellegrini

45

### Classe (1)

- **Définition conceptuelle** : une classe est l'implémentation d'un ou plusieurs types abstraits. C'est une définition statique de la structure et du comportement d'un ensemble d'objets.
- **Définition ensembliste** : une classe est un ensemble d'objets ayant exactement la même structure et le même comportement.

JP Domenger, G.Eyrolles, F.Pellegrini

46

### Classe (2)

- Classe ⇔ concrétisation d'un ou plusieurs types abstraits
  - Définit la structure et le comportement des objets instances de la classe
  - Réalise l'encapsulation en définissant la visibilité des constituants de l'objet : publique, privée, ...
  - La partie publique fournit une spécification (partielle) du ou des types abstraits

JP Domenger, G.Eyrolles, F.Pellegrini

47

### Classe (3)

- Une classe peut réaliser simultanément plusieurs rôles définis par des types abstraits différents
  - Implémentation des méthodes publiques correspondant aux opérations des différents types
  - Conflits à résoudre si opérations communes

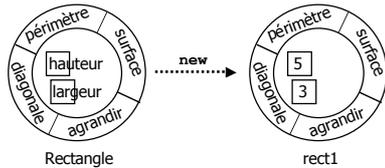
```
class Pointvoiture { // réalise Point + Voiture
    abscisse() // Abscisse du point
    ordonnée() // Ordonnée du point
    démarre() // Démarre la voiture
    arrête() // Arrête la voiture
    afficher() // Conflit entre représentations possibles
}
```

JP Domenger, G.Eyrolles, F.Pellegrini

48

### Classe (3)

- Un objet est une instance d'une unique classe



JP Domenger, G.Eyrolles, F Pellegrini

49

### Classe (4)

- Un objet peut tenir plusieurs rôles puisque une classe peut en implémenter plusieurs.
  - Exemple: La classe **FormeAnimée** implémente les types abstraits **Forme** et **Animé**. Les objets instances de cette classe peuvent être considérés soit comme des instances de **Forme** soit comme des instances d'**Animé**.

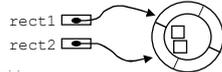
JP Domenger, G.Eyrolles, F Pellegrini

50

### Références (1)

- Une classe C définit un type c
- Une variable de type c peut faire référence à un objet de la classe C
- Deux variables différentes peuvent faire référence au même objet

```
Rectangle rect1; // Déclaration
Rectangle rect2;
rect1 = new Rectangle (3, 5); // Instanciation
rect2 = rect1; // Référence
```



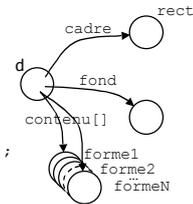
JP Domenger, G.Eyrolles, F Pellegrini

51

### Références (2)

- Les variables d'instances établissent des relations entre objets

```
class Dessin {
    Rectangle cadre;
    Couleur fond;
    Forme [] contenu;
}
...
d = new Dessin ();
d.cadre = rect;
d.fond = new Couleur ("Rouge");
d.contenu[0] = forme1;
d.contenu[1] = forme2;
...
```



JP Domenger, G.Eyrolles, F Pellegrini

52

### Identité et égalité

- Il y a identité d'objet lorsque deux références distinctes pointent sur le même objet
- Deux objets sont égaux s'ils sont dans le même état

```
rect1 = new Rectangle (3, 5);
rect2 = new Rectangle (6, 2);
rect3 = new Rectangle (3, 5);
rect4 = rect1
...
(rect1 == rect3) ⇨ false // Égalité des références
(rect1 == rect4) ⇨ true
rect1.equals (rect2) ⇨ false // Égalité des champs
rect1.equals (rect3) ⇨ true
rect1.equals (rect4) ⇨ true
```

JP Domenger, G.Eyrolles, F Pellegrini

53

### Propriétés de l'égalité

- L' égalité doit toujours vérifier les trois propriétés suivantes
  - Réflexivité : **a.equals (a)** doit toujours être vrai
  - Symétrie : **a.equals (b)** et **b.equals (a)** doivent toujours donner le même résultat
  - Transitivité : si **a.equals (b)** et **b.equals (c)** sont vrais, alors **a.equals (c)** doit toujours être vrai aussi

JP Domenger, G.Eyrolles, F Pellegrini

54

## Test d'égalité

- Le test d'égalité peut être :
  - Simple : comparaison des variables d'instance
  - Complexe : calcul sur les variables d'instance

```
class Fraction {
    int    numérateur;
    int    dénominateur;
    ...
    public boolean equals (Fraction f) {
        return ((this.numérateur * f.dénominateur) ==
                (this.dénominateur * f.numérateur));
    }
}
```

JP Domenger, G.Eyrolles, F Pellegrini

55

## Égalité de surface

- Il y a égalité de surface si on ne réalise qu'un test d'identité sur les champs des objets

```
((dess1.cadre == dess2.cadre)      &&
 (dess1.fond == dess2.fond)        &&
 (dess1.contenu[0] == dess2.contenu[0]) &&
 ...)
```

JP Domenger, G.Eyrolles, F Pellegrini

56

## Égalité profonde

- Il y a égalité profonde si on effectue (récursivement) des tests d'égalité

```
(dess1.cadre.equals(dess2.cadre)    &&
 dess1.fond.equals(dess2.fond)      &&
 dess1.contenu[0].equals(dess2.contenu[0]) &&
 ...)
```

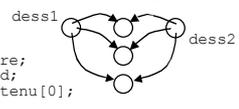
JP Domenger, G.Eyrolles, F Pellegrini

57

## Copie de surface

- Les considérations sur l'égalité s'appliquent aussi à la copie d'objets
- Il y a copie de surface si les champs de la copie font référence aux champs originaux, qui sont alors partagés par les deux objets

```
dess2.cadre = dess1.cadre;
dess2.fond  = dess1.fond;
dess2.contenu[0] = dess1.contenu[0];
...
```

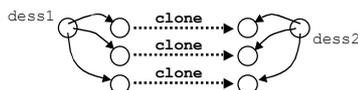


JP Domenger, G.Eyrolles, F Pellegrini

58

## Copie profonde

- Il y a copie profonde si les champs de la copie sont des clones des champs originaux



```
dess2 = new Dessin ();
dess2.cadre = dess1.cadre.clone ();
dess2.fond = dess1.fond.clone ();
dess2.contenu[0] = dess1.contenu[0].clone ();
...
dess2 = dess1.clone ();
```

JP Domenger, G.Eyrolles, F Pellegrini

59

## Interfaces, classes sous-typage et héritage de code.

**Comment les relations entre types et sous-types augmentent la modularité et la réutilisabilité**

## Interface (1)

- La notion de classe mélange spécification (*déclaration de type*) et implémentation.
- Nécessité de séparer les deux pour clarifier l'héritage.
  - Héritage d'interface (type, sous-type)
  - Héritage de code

JP Domenger, G.Eyrolles, F Pellegrini

61

## Interface (2)

- Interface ⇔ spécification de type abstrait
  - Ensemble de signatures d'opérations publiques ayant une sémantique commune
  - Aucune définition de code
- Classe ⇔ implémentation du type abstrait
  - Une classe peut réaliser le comportement de plusieurs interfaces, en définissant le code pour toutes les méthodes déclarées.

JP Domenger, G.Eyrolles, F Pellegrini

62

## Concrétisation

- Une classe peut réaliser une ou plusieurs interfaces

```
interface Vivant {
    void naît ();
    void meurt ();
    boolean estVivant ();
}
interface Mobile {
    void bouge (double x, double y);
}
class Lapin implements Vivant, Mobile {
    boolean coeurBat = false;
    void naît () { coeurBat = true; }
    void bouge (double x, double y) { ...
```

- La gestion des conflits dépend des langages

JP Domenger, G.Eyrolles, F Pellegrini

63

## Extension

- Une interface peut étendre une ou plusieurs autres interfaces

```
interface Mobile {
    void bouge (double x, double y);
}
interface Vivant {
    void naît ();
    void meurt ();
    boolean estVivant ();
}
interface Quadrupède extends Vivant, Mobile {
    void allonge (); // Extensions en plus de
    void debout (); // ce qui vient des deux
}
class Lapin implements Quadrupède { ...
```

JP Domenger, G.Eyrolles, F Pellegrini

64

## Hierarchie des types

- Classification des interfaces en fonction de leurs caractéristiques communes
- Relations de type *est-un* ou *est-une-sorte-de*
- Arborecence des interfaces par abstraction croissante
- Ordre partiel

JP Domenger, G.Eyrolles, F Pellegrini

65

## Sous-typage

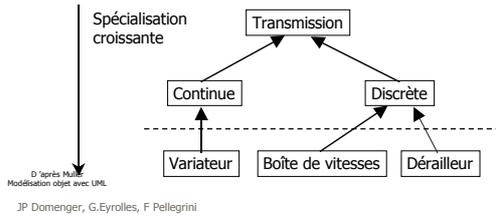
- Conformité des spécifications des interfaces
  - L'interface du type est toujours incluse dans l'interface du sous-type
- Principe de substitution
  - Il est possible de référencer une instance du sous-type en utilisant le sur-type. Une variable ou un paramètre de type  $\mathbb{T}$  peut donc indifféremment faire référence à un objet de type  $\mathbb{T}$  ou d'un sous-type de  $\mathbb{T}$

JP Domenger, G.Eyrolles, F Pellegrini

66

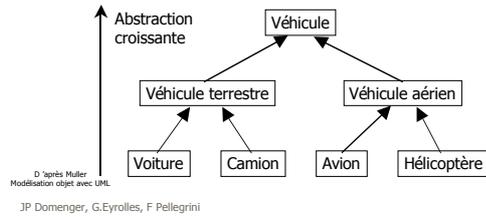
## Spécialisation

- Extension cohérente (par niveaux) d'un ensemble d'interfaces



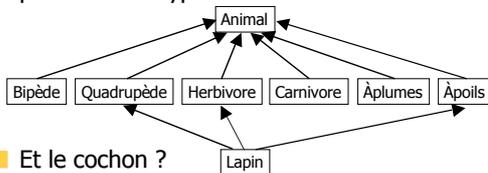
## Généralisation

- Un sur-type est une abstraction de ses sous-types



## Généralisation multiple

- Un sous-type peut être la spécialisation de plusieurs sur-types



- Et le cochon ?

## Héritage de code

- L'héritage (de code) permet :
  - La réutilisation de code existant (factorisation)
  - La spécialisation du code existant
  - La simplification du code par suppression des tests de typage, qui sont pris en charge par le système d'exécution en utilisant les liaisons dynamiques.

JP Domenger, G.Eyrolles, F.Pellegrini 70

## Sous-classe (1)

- La définition d'une sous-classe fille par héritage lui permet d'hériter du code de sa classe mère

```
class Position {
    double latitude;
    double longitude;
    double distance (Position p) {...}
    double distancePôleNord () {...}
}
class Géodésique extends Position {
    // Hérité: double latitude, longitude;
    // Hérité: double distance (Position p) {...}
    // Hérité: double distancePôleNord () {...}
    double altitude;
    double modifieAltitude (double alt) {...}
}
```

JP Domenger, G.Eyrolles, F.Pellegrini 71

## Sous-classe (2)

- Tout ce que peut faire la sur-classe, la sous-classe peut le faire. Mais elle peut le faire soit
  - en utilisant le code de la sur-classe.
  - en spécialisant le code de la sur-classe.

JP Domenger, G.Eyrolles, F.Pellegrini 72

## Redéfinition

- La sous-classe peut redéfinir des méthodes de sa classe mère, sauf avis contraire de celle-ci

```
class Géodésique extends Position {
    double altitude;
    double modifieAltitude (double alt) {...}
    double distance (Position p) {...}
    // La distance entre un point géodésique et une
    // position peut faire intervenir la hauteur du
    // premier. On spécialise le code de la sous-classe.
}
```

JP Domenger, G.Eyrolles, F Pellegrini

73

## Liaison dynamique

- Le choix de la méthode se fait par l'objet de manière dynamique à la réception d'un message.
  - Conservation du comportement de l'objet indépendamment du type qui le référence.

```
Animal a1, a2;
a1 = new Lapin (...);
a2 = new Tortue (...);
a1.bouge (); // Méthode bouge() de Lapin
a2.bouge (); // Méthode bouge() de Tortue
```

JP Domenger, G.Eyrolles, F Pellegrini

74

## Liaison statique

- Le choix de la méthode se fait statiquement en utilisant le type déclaré de l'objet durant la phase de compilation.
  - Non conservation du comportement de l'objet indépendamment du type qui le référence.

```
Lapin l = new Lapin (...);
Animal a = l;
l.bouge (); // Méthode bouge() de Lapin
a.bouge (); // Méthode bouge() de Animal
```

JP Domenger, G.Eyrolles, F Pellegrini

75

## Surcharge ≠ Redéfinition

- Utilisation du même nom mais avec des paramètres de types différents dans le même contexte (classe, fichier, application, ...)
  - Permet d'offrir le même service à partir de paramètres de types différents
  - Les méthodes de même nom doivent réaliser des opérations de même nature

```
afficher: Int I → Void
afficher: Char C → Void
afficher: Float F → Void
```

JP Domenger, G.Eyrolles, F Pellegrini

76

## Sélection de méthodes surchargées (1)

- La sélection d'une méthode surchargée se fait en fonction du type déclaré des paramètres et non en fonction des types réels.

```
class Animal{.....}
class Lapin extends Animal{.....}

class Elevage{
    public void nourrir(Lapin l){donner herbe}
    public void nourrir(Animal a){donner viande}
}
```

JP Domenger, G.Eyrolles, F Pellegrini

77

## Sélection de méthodes surchargées (2)

```
Elevage elev = new Elevage();
Lapin l = new Lapin();
Animal a = l; // a et l référence le même objet

elev.nourrir(a); // on lui donne de la viande
elev.nourrir(l); // on lui donne de l'herbe
```

JP Domenger, G.Eyrolles, F Pellegrini

78

## Surcharge

- La sous-classe peut surcharger des méthodes héritées de sa classe mère

```
class Géodésique extends Position {
    double altitude;
    double modifieAltitude (double alt) {...}

    // Hérité: double distance (Position p) {...}
    double distance (Géodésique g) {...}
    // La distance entre deux points géodésiques
    // peut faire intervenir leurs deux hauteurs
}
```

JP Domenger, G.Eyrolles, F Pellegrini

79

## Sélection de méthodes (1)

- En utilisant le type déclaré, on sélectionne une signature de méthode.
- En partant du type réel, on remonte jusqu'à trouver une méthode satisfaisante.

```
Position p = new Géodésique ();
Position q = new Géodésique ();

p.distance (q);
```

JP Domenger, G.Eyrolles, F Pellegrini

80

## Sélection de méthodes (2)

- En utilisant le type déclaré, on sélectionne une signature de méthode.
  - Signature = Position::distance(Position)
- En partant du type réel, on remonte jusqu'à trouver une méthode satisfaisante.
  - A partir de Géodésique
    - PositionGéodésique::distance(Position) n'existe pas
    - car toute les **Position** ne sont pas des **Géodésique**
  - On remonte à Position
    - on exécute Position::distance(Position)

JP Domenger, G.Eyrolles, F Pellegrini

81

## Héritage de code inutile

- On ne doit pas hériter de code et de données que l'on n'utilise pas

```
class PièceMécanique{ // Exemple à ne pas suivre
    String nom;
    int poids; // Poids de la pièce
    boolean estComposée; // Vrai si pièce composée
    int nombrePièces; // Nombre de sous-pièces
    Pièce [] composantes; // Tableau des composantes
    ...
    int poidsTotal () {
        if (estComposée) { // Test de type de pièce
            int pt = 0; // poids ne sert pas ici
            for (int i = 0; i < nombrePièces; i++)
                pt += composantes[i].poidsTotal ();
            return pt;
        } else return poids; // poids ne sert qu'ici
    }
}
```

JP Domenger, G.Eyrolles, F Pellegrini

82

## Méthode abstraite

- On ne doit factoriser que le code nécessaire à l'ensemble des sous-classes
- Les méthodes du type abstrait dont l'implémentation ne sera définie que dans les sous-classes sont déclarées comme abstraites dans la classe mère
- Une classe possédant au moins une méthode abstraite doit être déclarée abstraite elle aussi

JP Domenger, G.Eyrolles, F Pellegrini

83

## Classe abstraite

- Une classe déclarée comme abstraite ne peut jamais être instanciée
- Elle sert de conteneur à du code et des données qui seront utilisés par les sous-classes qui en hériteront
  - Uniformisation des noms
  - Factorisation du code

```
abstract class PièceMécanique {
    String nom; // Factorisation
    abstract int poidsTotal (); // Uniformisation
}
```

JP Domenger, G.Eyrolles, F Pellegrini

84

## Implémentation

- Les sous-classes implémentent les méthodes abstraites de leur classe mère

```
class PièceMécaniqueSimple extends Pièce {
    int poids; // Poids de la pièce
    int poidsTotal () { // Implémentation
        return poids;
    }
}
class PièceMécaniqueComposée extends Pièce {
    int nombrePièces; // Nombre de sous-pièces
    Pièce [] composantes; // Tableau des composantes
    int poidsTotal () { // Implémentation
        int pt = 0;
        for (int i = 0; i < nombrePièces; i ++)
            pt += composantes[i].poidsTotal ();
        return pt;
    }
}
```

JP Domenger, G.Eyrolles, F Pellegrini

85

## Héritage multiple

- Possibilité de faire hériter une classe du code de plusieurs sur-classes
- Complexité de la résolution des conflits en cas d'identité de noms de méthode
- On n'hérite souvent que du code d'une seule classe, les autres étant purement abstraites (□ interfaces)

JP Domenger, G.Eyrolles, F Pellegrini

86

## Héritage ou délégation

- L'héritage peut engendrer une prolifération de classe, surtout si il y a composition de comportement (*cas de l'héritage multiple*).
- La délégation est une alternative souple à l'héritage.

JP Domenger, G.Eyrolles, F Pellegrini

87

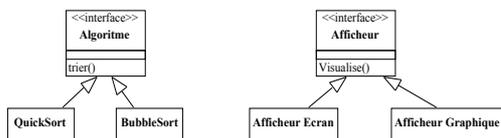
## Exemple (1)

- On dispose d'une hiérarchie d'algorithmes de tri. On dispose aussi d'une hiérarchie d'afficheurs. On veut pouvoir composer un algorithme de tri avec un afficheur.

JP Domenger, G.Eyrolles, F Pellegrini

88

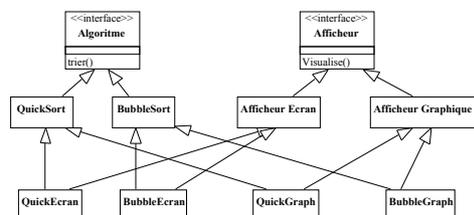
## Exemple(2)



JP Domenger, G.Eyrolles, F Pellegrini

89

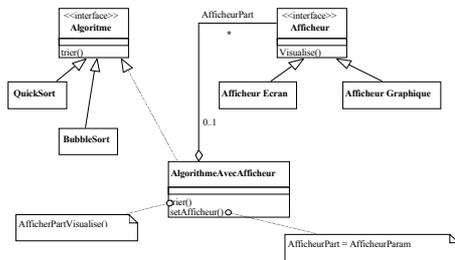
## Exemple(3) : Version héritage



JP Domenger, G.Eyrolles, F Pellegrini

90

### Exemple(3) : Version avec délégation



JP Domenger, G.Eyrolles, F Pellegrini

91

### Polymorphisme

**Le véritable intérêt de l'orienté objet.**

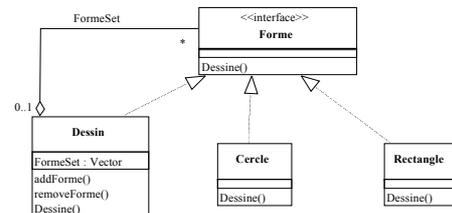
### Polymorphisme

- Utilisation des relations types/sous-types pour obtenir l'unification des différentes classes.
- Utilisation des liaisons dynamiques pour conserver la spécificité du comportement.

JP Domenger, G.Eyrolles, F Pellegrini

93

### Polymorphisme (exemple)



JP Domenger, G.Eyrolles, F Pellegrini

94

### Polymorphisme (exemple)

- Forme est une interface qui admet plusieurs réalisations. On dit que Forme est **polymorphe**.
- Cercle et Rectangle sont des classes qui réalisent l'interface Forme.
- Dessin est une classe composite qui contient des Forme et qui est une Forme.

JP Domenger, G.Eyrolles, F Pellegrini

95

### Le code de Dessin

```

Class Dessin implements Forme{
    private Vector FormeSet;

    public void addForme (Forme f){
        FormeSet.AddElement(f);
    }
    public void removeForme(Forme f){
        FormeSet.removeElement(f);
    }
    public void Dessine(){
        for(int i = 0; i < FormeSet.size(); i++)
            ((Forme) FormeSet.elementAt(i)).dessine();
    }
}
    
```

JP Domenger, G.Eyrolles, F Pellegrini

96

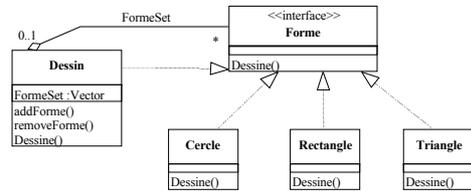
## Le code de l'application

```
class Main {
    static public void Main(String [] argv){
        Dessin d = new Dessin();
        d.addForme(new Cercle());
        d.addForme(new Rectangle());
        .....
    }
}
```

JP Domenger, G.Eyrolles, F Pellegrini

97

## Extension avec un triangle



JP Domenger, G.Eyrolles, F Pellegrini

98

## Le nouveau code de l'application

```
class Main {
    static public void Main(String [] argv){
        Dessin d = new Dessin();
        d.addForme(new Cercle());
        d.addForme(new Rectangle());
        d.addForme(new Triangle());
        .....
    }
}
```

JP Domenger, G.Eyrolles, F Pellegrini

99

## Les techniques de réutilisation

**Intérêt de l'héritage et du polymorphisme.**

## Technique de Réutilisation

- Critère
  - Séparation de l'interface et de l'implémentation
- Différentes techniques
  - Boîte Noire : aucune adaptation à faire, pas de connaissance de l'implémentation
  - Boîte Grise : adaptation au contexte par paramétrisation, pas de connaissance du source

JP Domenger, G.Eyrolles, F Pellegrini

101

## Technique de réutilisation

- Autres techniques
  - Boîte blanche : modification du code source, afin de l'adapter au contexte
    - Boîte en verre : consulter le code (l'implémentation) sans le modifier

JP Domenger, G.Eyrolles, F Pellegrini

102

## Les techniques d'extension et de réutilisation

- Boîtes Noires
  - Héritage public d'interface
- Boîtes Grise
  - Les poignées
  - La délégation

JP Domenger, G.Eyrolles, F Pellegrini

103

## Héritage public

```
class AlgorithmeTri
{
    virtual void Tri()=0
};
class AlgorithmeTriAffiche : public AlgorithmeTri
{
    private void afficheEtape(){.....}
    virtual void Tri() {
        .....
        afficheEtape (...);
    }
}
```

*Peut engendrer de la duplication de code*

JP Domenger, G.Eyrolles, F Pellegrini

104

## Les poignées

```
Class AlgorithmeTri {
    protected:
        virtual void poignéeAffiche() {}
    public
        void affiche() {..... poignéeAffiche();.....}
}
class AlgorithmeTriAffiche:public AlgorithmeTri{
    protected:
        virtual void poignéeAffiche() {.....}
```

*On diminue la duplication de code, mais le nombre de classe augmente.  
Il est nécessaire de prévoir toutes les extensions de la classe lors de la conception*

JP Domenger, G.Eyrolles, F Pellegrini

105

## La délégation

```
class AlgorithmeTri {
    Afficheur *affiche;
    public:
        void Tri() { ..... affiche->visualise();}
};
void setAfficheur(Afficheur *aff) { affiche = aff;}
class Afficheur{
    public:
        virtual void visualise(){.....}
```

*On minimise la duplication de code et le nombre de classes*

JP Domenger, G.Eyrolles, F Pellegrini

106

## Exercices

- Un analyseur doit lire un message et déclencher l'action associée au message
  - Connect --> lancer une nouvelle connexion.
  - Disconnect --> arrêter la connexion.
- Un nouveau message doit être pris en compte
  - Print --> affiche l'état des connexions

JP Domenger, G.Eyrolles, F Pellegrini

107

## Héritage simple (1)

```
class Parser
{
    virtual boolean parse (String s){
        if( s==« Connect »){
            Connexion::NewConnexion();
            return true;
        }
        if(s==« Disconnect »){
            Connexion::DetruireConnexion();
            return true;
        }
        return false;
    }
};
```

JP Domenger, G.Eyrolles, F Pellegrini

108

## Héritage simple (2)

```
class ParserPrint : public Parser
{
    virtual boolean parse(String s) {
        if(s==« Print ») {
            Connexion::Print();
            return true;
        }
        return Parser::parse(s);
    }
};
```

JP Domenger, G.Eyrolles, F Pellegrini

109

## Poignée (1)

```
class Parser
{
    virtual protected boolean handle(String s){
        return false;
    }
    virtual boolean parse (String s){
        if( s==« Connect »){
            Connexion::NewConnexion();
            return true;
        }
        if(s==« Disconnect »){
            Connexion::DetruireConnexion();
            return true;
        }
        return handle(s);
    }
}
```

JP Domenger, G.Eyrolles, F Pellegrini

110

## Poignée (2)

```
class ParserPrint : public Parser
{
    virtual boolean handle (String s) {
        if(s==« Print ») {
            Connexion::Print();
            return true;
        }
        return false;
    }
};
```

JP Domenger, G.Eyrolles, F Pellegrini

111

## Délégation (1)

```
Class ParserInterface // une interface en C++ {
    virtual boolean parse(String S) = 0;
}
class ParserDefault: public ParserInterface {
    virtual boolean parse (String s){
        if( s==« Connect »){
            Connexion::NewConnexion();
            return true;
        }
        if(s==« Disconnect »){
            Connexion::DetruireConnexion();
            return true;
        }
        return false;
    }
}
```

JP Domenger, G.Eyrolles, F Pellegrini

112

## Délégation (2)

```
class ParserDecorateur: public ParserInterface {
private:
    ParserInterface component;
public:
    ParserDecorateur(ParserInterface pi){
        component = pi;
    }
    virtual boolean parser (String s) {
        return component.parser(s);
    }
};
```

JP Domenger, G.Eyrolles, F Pellegrini

113

## Délégation (3)

```
class ParserPrint: public ParserDecorateur
{
public:
    ParserPrint(ParserInterface pi):ParserDecorateur(pi)
    {};
    virtual boolean parser (String s) {
        if(s==« Print ») {
            Connexion::Print();
            return true;
        }
        return ParseDecorateur::parser(s);
    }
}
```

JP Domenger, G.Eyrolles, F Pellegrini

114

## Approfondissement de l'héritage

**Les relations entre:**

- 1- interfaces**
- 2- classes**
- 3- classes et interfaces**

## Spécification/Implémentation

- Interface : type uniquement
- Classe : ambigu, à la fois le type et le code
- Héritage public : ambigu, à la fois type et code
- Héritage privé : non-ambigu, uniquement le code

JP Domenger, G.Eyrolles, F Pellegrini 116

## Relations statiques

- Relation entre interfaces.
  - Héritage public (généralisation/spécialisation)
- Relation interface/classe
  - Concrétisation
- Relation classes/classes
  - Héritage privé

JP Domenger, G.Eyrolles, F Pellegrini 117

## Relations entre Interface

```

classDiagram
    class Graphic {
        <<interface>>
        -Graphic
        **paint()
    }
    class Runnable {
        <<interface>>
        -Runnable
        **run()
        **stop()
    }
    class GraphicRunnable {
        <<interface>>
        -GraphicRunnable
    }
    GraphicRunnable --|> Graphic
    GraphicRunnable --|> Runnable
    
```

JP Domenger, G.Eyrolles, F Pellegrini 118

## Relations entre interfaces

- L'interface GraphicRunnable peut tenir les rôles de Graphic et de Runnable

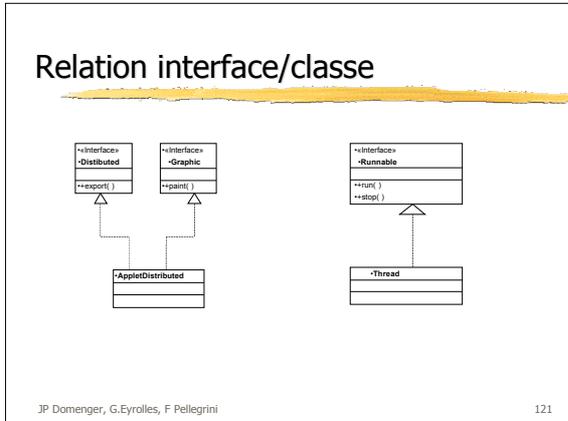
JP Domenger, G.Eyrolles, F Pellegrini 119

## Relations entre interfaces en C++

```

class Graphic
{
public:
    virtual void paint() = 0;
};
class Runnable
{
public:
    virtual void run() = 0;
    virtual void stop() = 0;
};
class GraphicRunnable: public Graphic, public Runnable
{}
    
```

JP Domenger, G.Eyrolles, F Pellegrini 120



### Relation interface/classe

- La classe AppletDistributed concrétise les classe Graphic et Distributed
- La classe Thread concrétise la classe Runnable

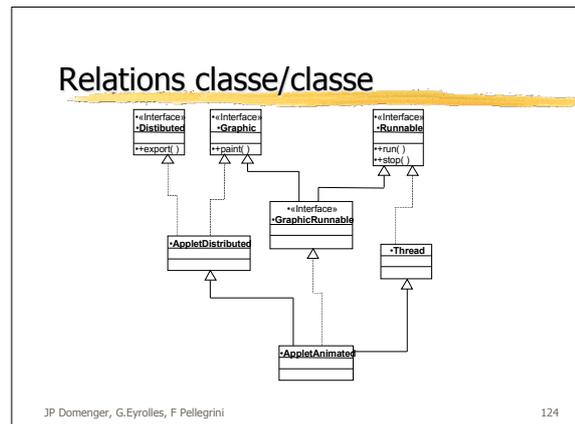
JP Domenger, G.Eyrolles, F Pellegrini 122

### Relation interface/classe en C++

```

class Distributed
{public:
    virtual void export() = 0;
};
class AppletDistribué:public Applet, public Distributed
{ public:
    virtual void paint(){ implémentation de paint };
    virtual void export(){ implémentation de export};
};
class Thread: public Runnable
{public:
    virtual void run() { implémentation de run };
    virtual void stop() { implémentation de stop };
};
    
```

JP Domenger, G.Eyrolles, F Pellegrini 123



### Relations classe/classe

- La classe AppletAnimated hérite des classes AppletDistributed et de Thread

*Le problème est que les instances de AppletAnimated peuvent être distribuées. Comme l'héritage est une relation publique, par transitivité, on obtient des comportements supplémentaires non pertinents.*

JP Domenger, G.Eyrolles, F Pellegrini 125

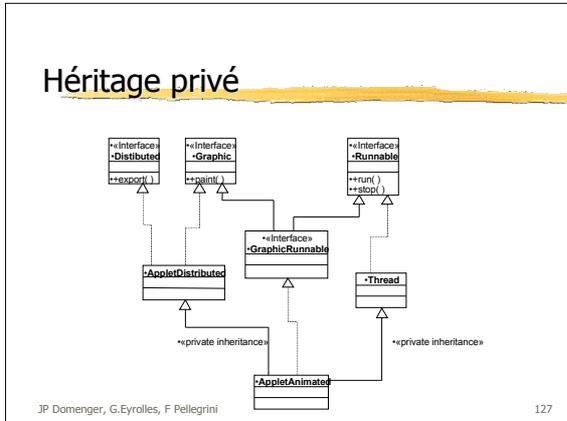
### Traduction en C++

```

Class AppletAnimated: public GraphicRunnable,
                    public AppletDistributed, public Thread
{
};

void function()
{
    Distributed *dis = new AppletAnimated();
}
    
```

JP Domenger, G.Eyrolles, F Pellegrini 126



### Héritage privé

- Si on ne veut que le code, et pas le type il faut utiliser l'héritage privée.
- Relation entre classes, donc statique

JP Domenger, G.Eyrolles, F Pellegrini 128

### Héritage privée en C++

```

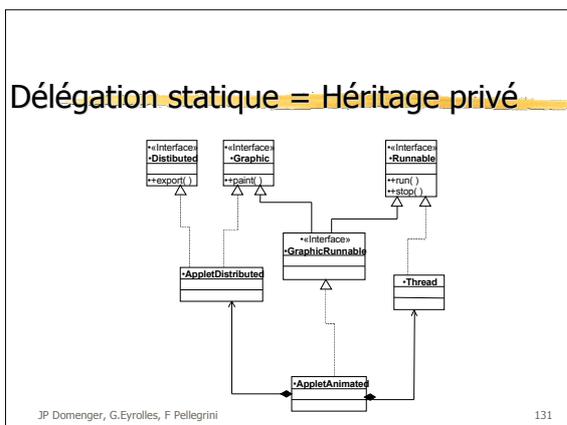
class AppletAnimated: public GraphicRunnable,
                    private AppletDistributed,
                    private Thread
{
public:
    virtual void paint() {AppletDistributed::paint();}
    virtual void run () {Thread::run();}
    virtual void stop () {Thread::stop();}
};
void function()
{
    Distributed *dis = new AppletAnimated();
    // erreur de compilation.
}
    
```

JP Domenger, G.Eyrolles, F Pellegrini 129

### Délégation = Héritage privé

- Si on ne veut que le code, et pas le type il faut utiliser la délégation.
- Délégation statique
- Délégation dynamique

JP Domenger, G.Eyrolles, F Pellegrini 130



### Délégation statique en C++

```

class AppletAnimated: public GraphicRunnable,
{
    private:
        AppletDistributed appletDelegated;
        Thread threadDelegated;
    public:
        virtual void paint() {appletDelegated.paint();}
        virtual void run () {threadDelegated.run();}
        virtual void stop () {threadDelegated.stop();}
};
    
```

JP Domenger, G.Eyrolles, F Pellegrini 132

## Délégation dynamique en C++

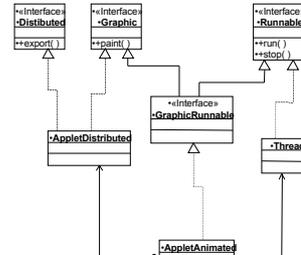
```

Class AppletAnimated: public GraphicRunnable,
{
private:
    Graphic *graphicDelegated;
    Runnable * threadDelegated;
public:
    AppletAnimated(Graphic *g,Runnable *r)
    {
        graphicDelegated = g;
        runnableDelegated = r;
    }
    virtual void paint () {appletDelegated->paint ()};
    virtual void run () {threadDelegated->run ()};
    virtual void stop () {threadDelegated->stop ()};
};
    
```

JP Domenger, G.Eyrolles, F Pellegrini

133

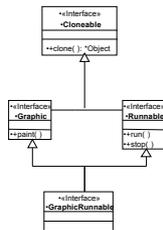
## Le diagramme du code précédent



JP Domenger, G.Eyrolles, F Pellegrini

134

## Une modification du schéma



JP Domenger, G.Eyrolles, F Pellegrini

135

## Délégation dynamique en C++

```

Class AppletAnimated: public GraphicRunnable,
{private:
    Graphic *graphicDelegated;
    Runnable * threadDelegated;
public:
    AppletAnimated(Graphic g,Runnable r)
    {
        graphicDelegated = g.clone();
        runnableDelegated = r.clone();
    }
    virtual void paint () {appletDelegated->paint ()};
    virtual void run () {threadDelegated->run ()};
    virtual void stop () {threadDelegated->stop ()};
};
    
```

JP Domenger, G.Eyrolles, F Pellegrini

136

## La réutilisation

- Réutilisation d 'architecture (framework)
- Réutilisation de composants d'architecture (pattern)
- Réutilisation de rôles (interface)
- Réutilisation de classes (bibliothèque)

JP Domenger, G.Eyrolles, F Pellegrini

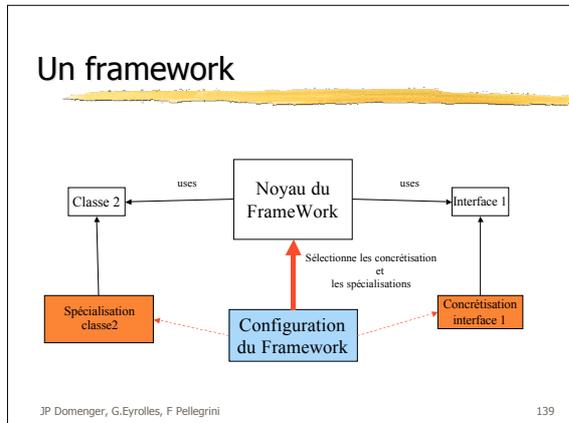
137

## Framework

- Framework
  - Un ensemble coopérant de classes, certaines peuvent étre abstraites, cet ensemble est réutilisable pour certains types de problème.

JP Domenger, G.Eyrolles, F Pellegrini

138



### Patterns

- Proposer une micro architecture pour résoudre un problème donné.
  - Les patterns sont les principaux composants des frameworks.
  - Créationnels, Comportementaux, Structuraux

JP Domenger, G.Eyrolles, F Pellegrini 140

### Pattern

■ L'initialisation est une abstract factory.

```

Class Configuration
{
    Classe2* creerClasse2()
    {
        return new SpécialisationClasse2();
    }
    Interfacel* creerInterfacel()
    {
        return new ConcretisationInterfacel();
    }
}
    
```

JP Domenger, G.Eyrolles, F Pellegrini 141

### Une autre implémentation !

```

class Cloneable{
public:
    virtual Cobject* clone() = 0;
};
class Classe2: public Cloneable {
public:
    virtual Cobject* clone(){
        // allocation memoire spécifique;
        this-> StateCopy();
    }
protected:
    void StateCopy(){ // Copie de l'état
    };
}
    
```

JP Domenger, G.Eyrolles, F Pellegrini 142

### Une autre implémentation !

```

class SpécialisationClasse2: public Classe2
{
public:
    virtual Cobject* clone(){
        SpécialisationClasse2 *tmp=new SpécialisationClasse2();
        this.StateCopy();
    }
protected:
    void StateCopy(){ Classe2::StateCopy();.....}
};
    
```

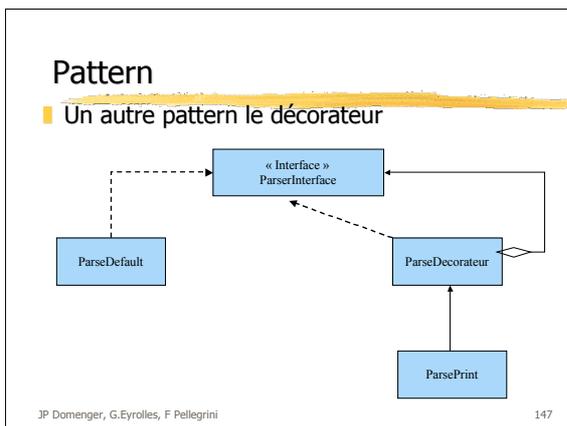
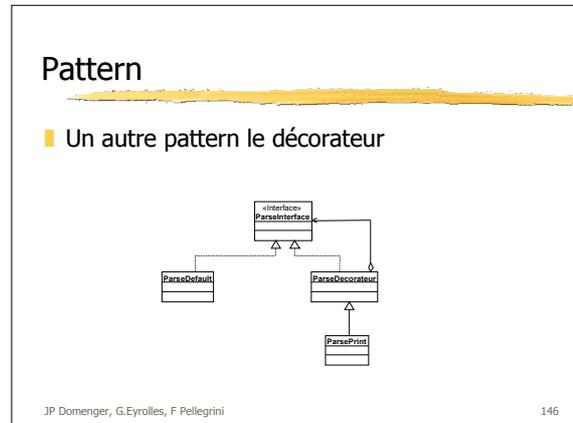
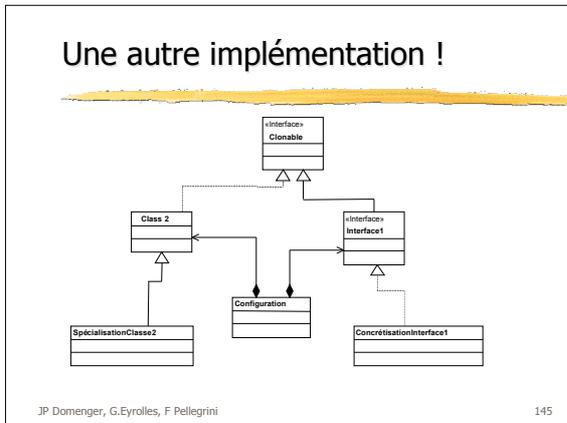
JP Domenger, G.Eyrolles, F Pellegrini 143

### Une autre implémentation !

```

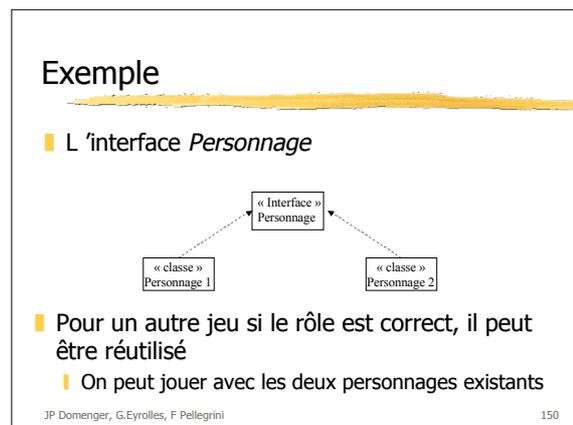
class Configuration {
private:
    Classe2 *protoClasse2;
    Interfacel *protoInterfacel;
public:
    Configuration (Classe2 *c, Interfacel *i){
        protoClasse2 = c;
        protoInterfacel = i;
    }
    Classe2* creerClasse2() {
        return (dynamic_cast<Classe2 *>) c.clone();
    }
    Interfacel* creerInterfacel() {
        return (dynamic_cast<Interfacel *>) i.clone();
    }
}
    
```

JP Domenger, G.Eyrolles, F Pellegrini 144



- ### Relations
- Relation classes/classes
    - Héritage privé/ Délégation
    - Réutilisation de code
  - Relation interface/classe
    - Concrétisation
    - Adaptation d'application
  - Relation entre interfaces.
    - Héritage public (généralisation/spécialisation)
    - Réutilisation de partie d'application par une nouvelle application
- JP Domenger, G.Eyrolles, F Pellegrini 148

- ### Concevoir pour réutiliser
- Définir des rôles qui soient réutilisables
    - indépendant de l'application
    - représentant correct de l'abstraction
    - interface complète et minimale
- Favoriser la réutilisation en phase d'analyse.  
Permettre de conserver les réalisations de rôles*
- JP Domenger, G.Eyrolles, F Pellegrini 149



## Concevoir pour réutiliser

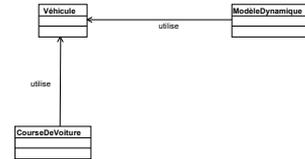
- Identifier clairement les relations de dépendance entre les composants du framework et les types
  - les sous-types bénéficient alors du composant dépendant du sur-type et de lui seul.

*Diminue le couplage entre composants  
Favorise la réutilisabilité*

JP Domenger, G.Eyrolles, F Pellegrini

151

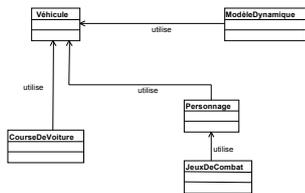
## Exemples



JP Domenger, G.Eyrolles, F Pellegrini

152

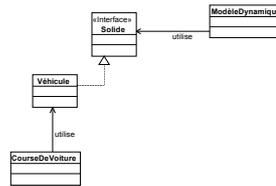
## Exemples



JP Domenger, G.Eyrolles, F Pellegrini

153

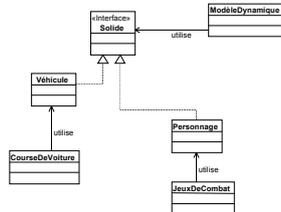
## Exemples



JP Domenger, G.Eyrolles, F Pellegrini

154

## Exemples



JP Domenger, G.Eyrolles, F Pellegrini

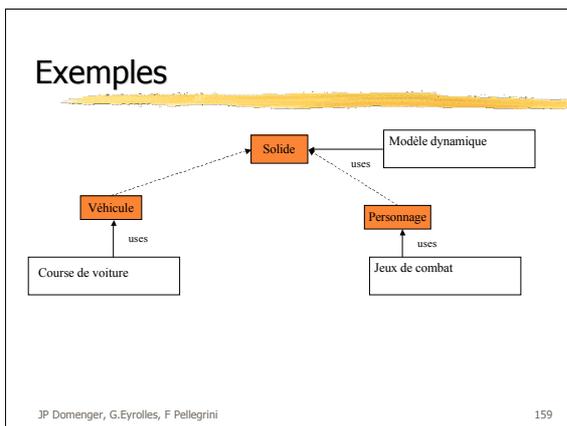
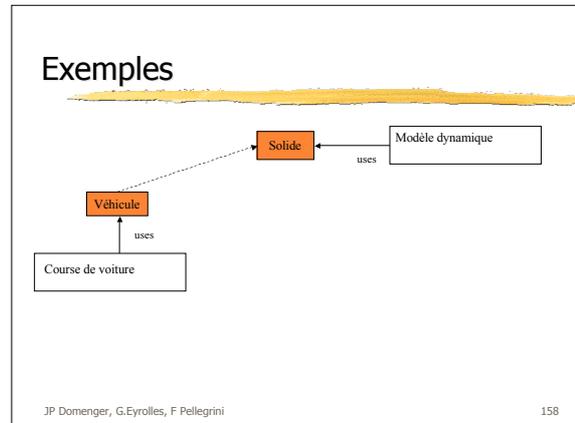
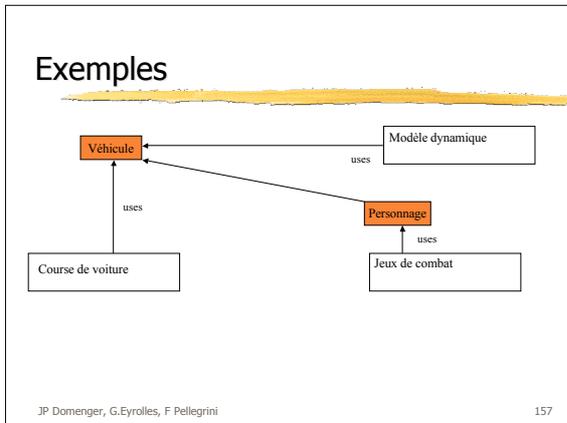
155

## Exemples



JP Domenger, G.Eyrolles, F Pellegrini

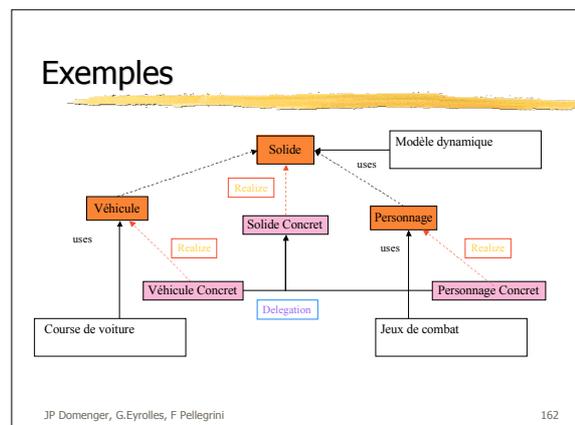
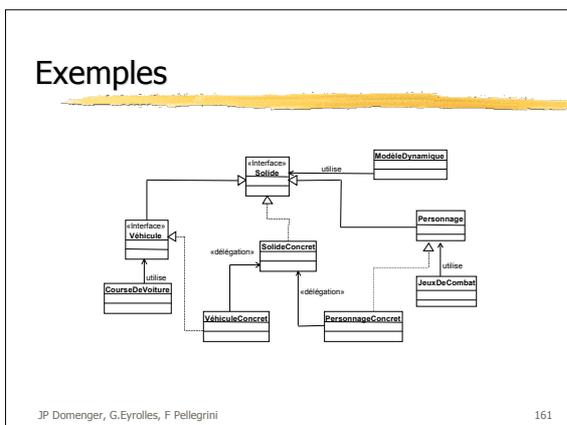
156



### Attention aux Ayatollah du type

- Sur cette exemple, il y a nécessairement de la duplication de code.
- Encapsulation pure et dure
  - Perte d'efficacité

JP Domenger, G.Eyrolles, F Pellegrini 160



## Concevoir pour réutiliser

- Prévoir des mécanismes pour insérer facilement de nouvelle réalisation d'interfaces
  - Séparation entre l'instanciation de classes et son utilisation
    - patterns créationels (Abstract factory, ....)

*Favorise l'extension d 'application  
Adaptation de framework*

JP Domenger, G.Eyrolles, F Pellegrini

163

## Concevoir pour réutiliser

- Prévoir des mécanismes pour pouvoir paramétrer le comportement d'une classe
  - Poignée
  - Délégation du comportement à un objet concrétisant un rôle (Pattern comportementaux)

*Favorise l'adaptation de classe*

JP Domenger, G.Eyrolles, F Pellegrini

164