Université Nice Sophia Antipolis

Patron de Comportement: Visiteur

F. Mallet

miage.m1@gmail.com

http://deptinfo.unice.fr/~fmallet/

Université Nice Sophia Antipolis

Motivation

- Les besoins pour une bonne conception et du bon code :
  - Extensibilité
  - Flexibilité
  - Facilité à maintenir
  - Réutilisabilité
  - Les qualités internes
  - Meilleure spécification, construction, documentation

Université Nice Sophia Antipolis

Historique

- MVC
- Gang of Four : Erich Gamma, Richard Helm,Ralph Johnson, John Vlissides
  - Définition de 23 patterns
- Design Patterns – Elements of Reusable Object-Oriented Software, Addison Wesley, 1994
- Un Design Pattern nomme, abstrait et identifie les aspects essentiels d'une structuration récurrente, ce qui permet de créer une modélisation orientée objet réutilisable

Université Nice Sophia Antipolis

Classification

- Création
  - Comment un objet peut être créé
  - Indépendance entre la manière de créer et la manière d'utiliser
- Structure
  - Comment les objets peuvent être combinés
  - Indépendance entre les objets et les connexions
- Comportement
  - Comment les objets communiquent
  - Encapsulation de processus (ex : observer/observable)

Université Nice Sophia Antipolis

Visiteur

- Intention
  - Représenter **UNE** opération à effectuer sur les éléments d'une structure
  - Permet de définir une nouvelle opération sans changer les classes des éléments sur lesquels on opère
- Exemple
  - Un arbre de syntaxe abstraite pour un compilateur, un outil XML…Différents traitement sur le même arbre : type check, optimisation, analyses…
  - Faire la somme des jours de congés des employés d'une entreprise
  - Recensement / *pretty-print* des bébêtes

Université Nice Sophia Antipolis

Application

- Champs d'application
  - Une structure contient beaucoup de classes aux interfaces différentes
  - Pour éviter la pollution des classes de la structure
  - Les classes définissant la structure changent peu, mais de nouvelles opérations sont toujours nécessaires

Sans Visiteur

- Exemple

```
interface Figure {
  void prettyPrint(String tab);  // autant de méthode publiques que de services
  void draw(Graphics g);         // dépendence sur AWT
}

class Composite implements Figure {
  Figure[] children;

  public void prettyPrint(String tab) {
    tab += "\t";
    for(Figure child : children)
      child.prettyPrint(tab);
  }
  public void draw(Graphics g) {
    for(Figure child : children)
      child.draw(g);
  }
}
```

Sans Visiteur

- Exemple

```
interface Figure {
  void prettyPrint(String tab);  // autant de méthode publiques que de services
  void draw(Graphics g);         // dépendence sur AWT
}

class Circle implements Figure {
  double x, y, r;

  public void prettyPrint(String tab) {
    System.out.println(tab+"Circle:"+this);
  }
  public void draw(Graphics g) {
    g.drawEllipse(x, y, r*2, r*2);
  }
}
```

Sans Visiteur

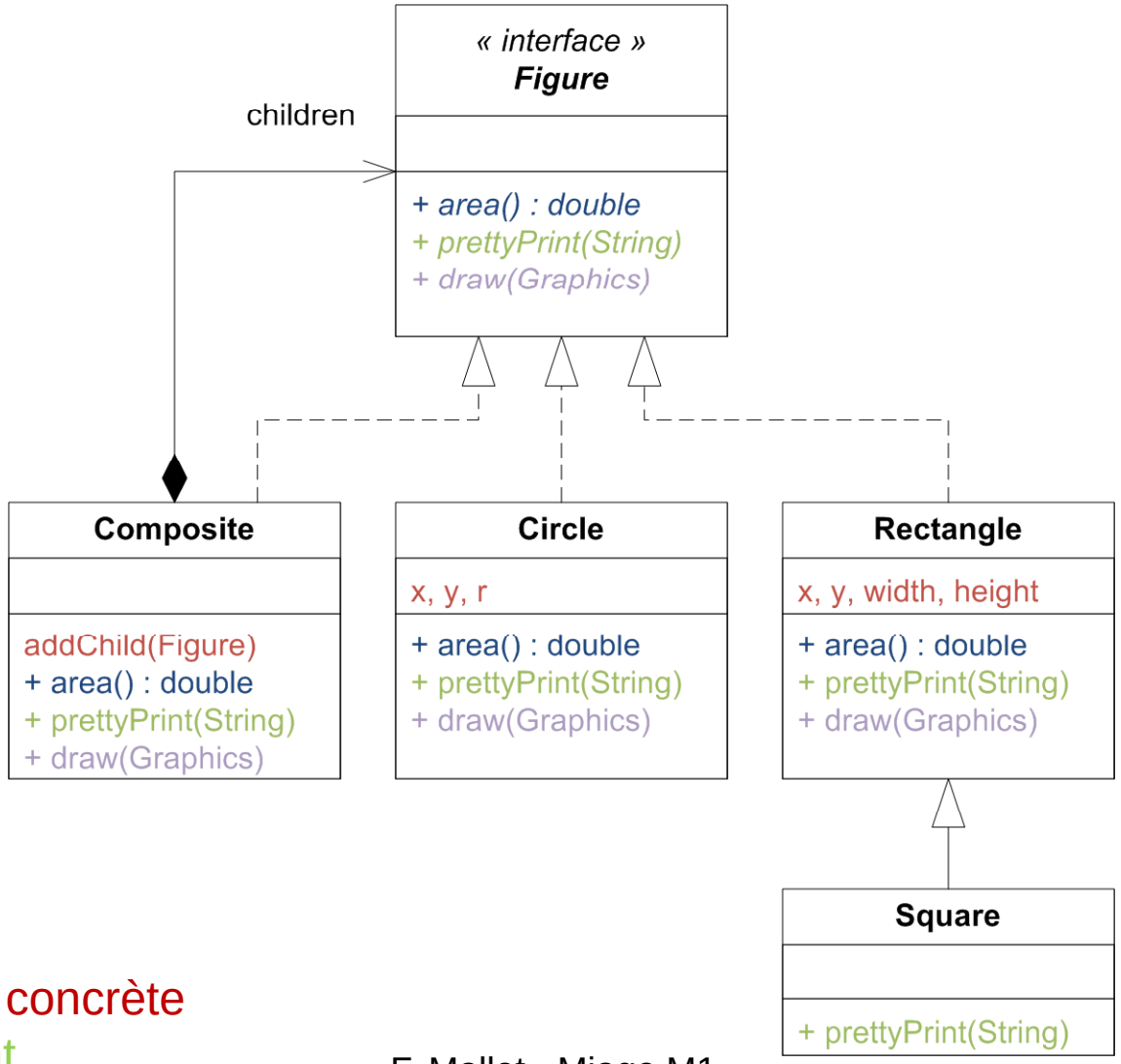
```
interface Figure {
  void prettyPrint(String tab);  // autant de méthode publiques que de services
  void draw(Graphics g);         // dépendence sur AWT
}

class Rectangle implements Figure {
  double x, y, width, height;

  public void prettyPrint(String tab) {
    System.out.println(tab+"Rectangle:"+this);
  }
  public void draw(Graphics g) {
    g.drawRect(x, y, width, height);
  }
}

class Square extends Rectangle {
  public void prettyPrint(String tab) {
    System.out.println(tab+"Square:"+this);
  } // hérite de la méthode draw
}
```

Figures sans visiteurs



Métier Figure
Spécifique classe concrète
Service PrettyPrint
Service draw

Université Nice Sophia Antipolis

Avec Visiteur

- Exemple

```
interface Figure {
  void accept(FigureVisitor visitor);
}
interface FigureVisitor {
  void visit(Composite c);
  void visit(Circle c);
  void visit(Rectangle r);
  void visit(Square s);
}

class Composite implements Figure, Iterable<Figure> {
  Figure[] children;

  public void accept(FigureVisitor visitor) {
    visitor.visit(this);
  }
}
```

Avec Visiteur

- Exemple

```
interface Figure {
  void accept(FigureVisitor visitor);
}

class Circle implements Figure {
  double x, y, r;

  public void accept(FigureVisitor visitor) {
    visitor.visit(this);
  }
}
```

Avec Visiteur

- Exemple

```
interface Figure {
  void accept(FigureVisitor visitor);
}

class Rectangle implements Figure {
  double x, y, width, height;

  public void accept(FigureVisitor visitor) {
    visitor.visit(this);
  }
}

class Square implements Figure {
  Square(double x, double y, double length) {
    super(x, y, length, length);
  }
  public void accept(FigureVisitor visitor) {
    visitor.visit(this);
  }
}
```

Université Nice Sophia Antipolis

PrettyPrint

Exemple

```
class PrettyPrint implements FigureVisitor {
  String tab = "";
  public void visit(Composite c) {
    String oldTab = tab;
    tab += "\t";
    for(Figure child : c)
      child.accept(this);
    tab = oldTab;
  }
  public void visit(Circle c) {
    System.out.println(tab + "Circle:" + c);
  }
  public void visit(Rectangle r) {
    System.out.println(tab + "Rectangle:" + r);
  }
  public void visit(Square s) {
    System.out.println(tab + "Square:" + r);
  }
}
```

Dessiner dans un Graphics

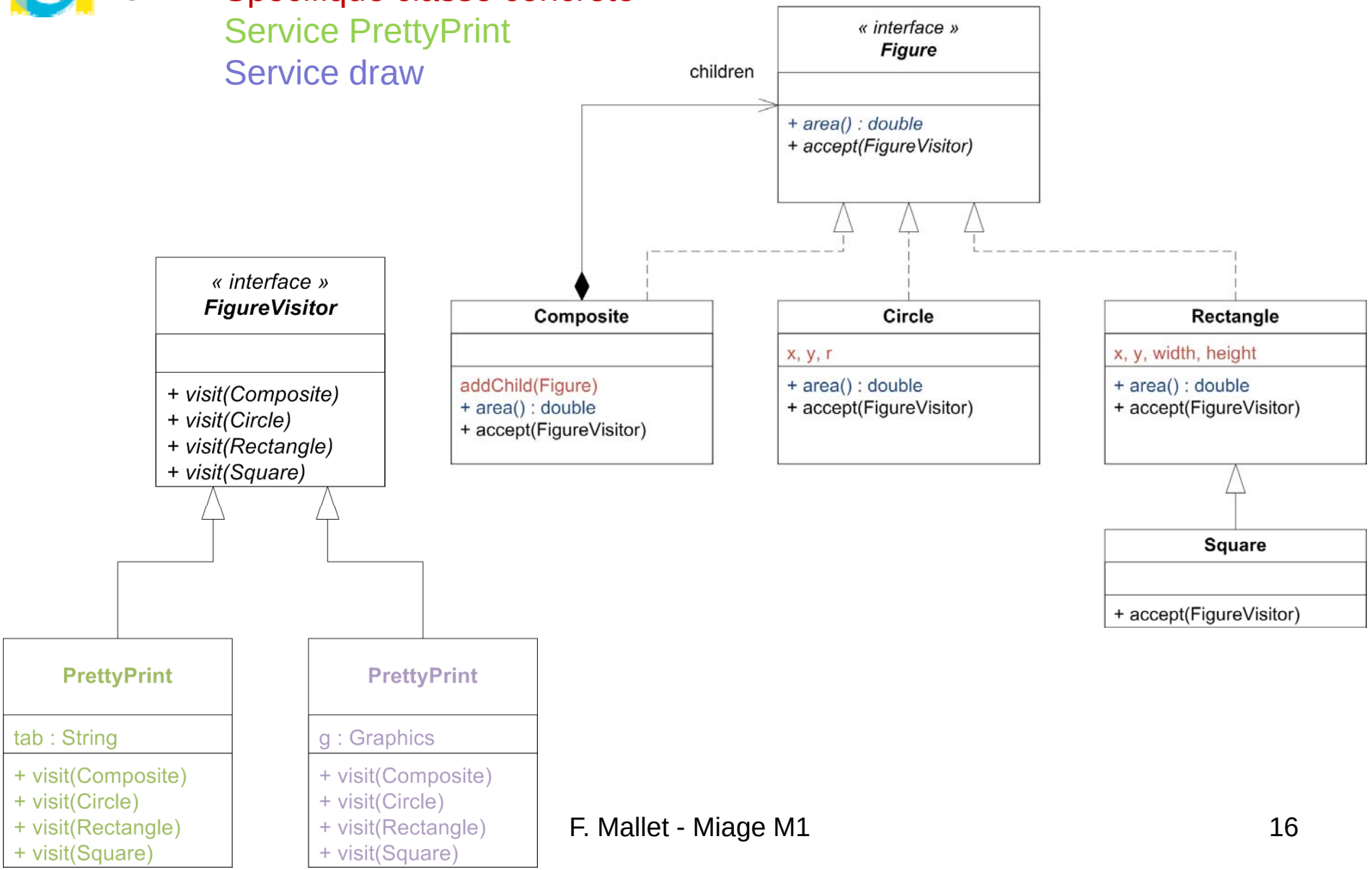
- Exemple

```
class FigureDrawer implements FigureVisitor {
  Graphics g;
  FigureDrawer(Graphics g) { this.g = g; }

  public void visit(Composite c) {
    for(Figure child : c)
      child.accept(this);
  }
  public void visit(Circle c) {
    g.drawEllipse(c.x, c.y, c.r*2, c.r*2);
  }
  public void visit(Rectangle r) {
    g.drawRect(r.x, r.y, r.width, r.height);
  }
  public void visit(Square s) {
    this.visit((Rectangle)s);
  }
}
```

Figures avec visiteurs

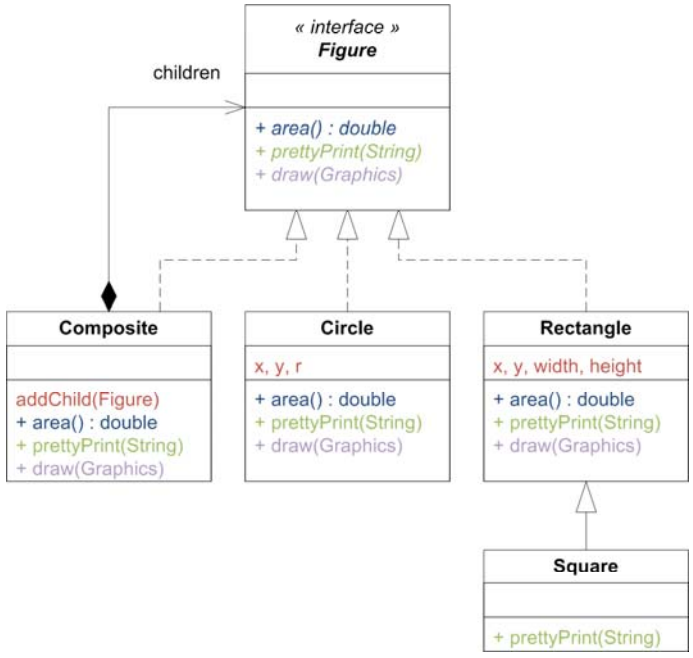
Métier Figure
Spécifique classe concrète
Service PrettyPrint
Service draw



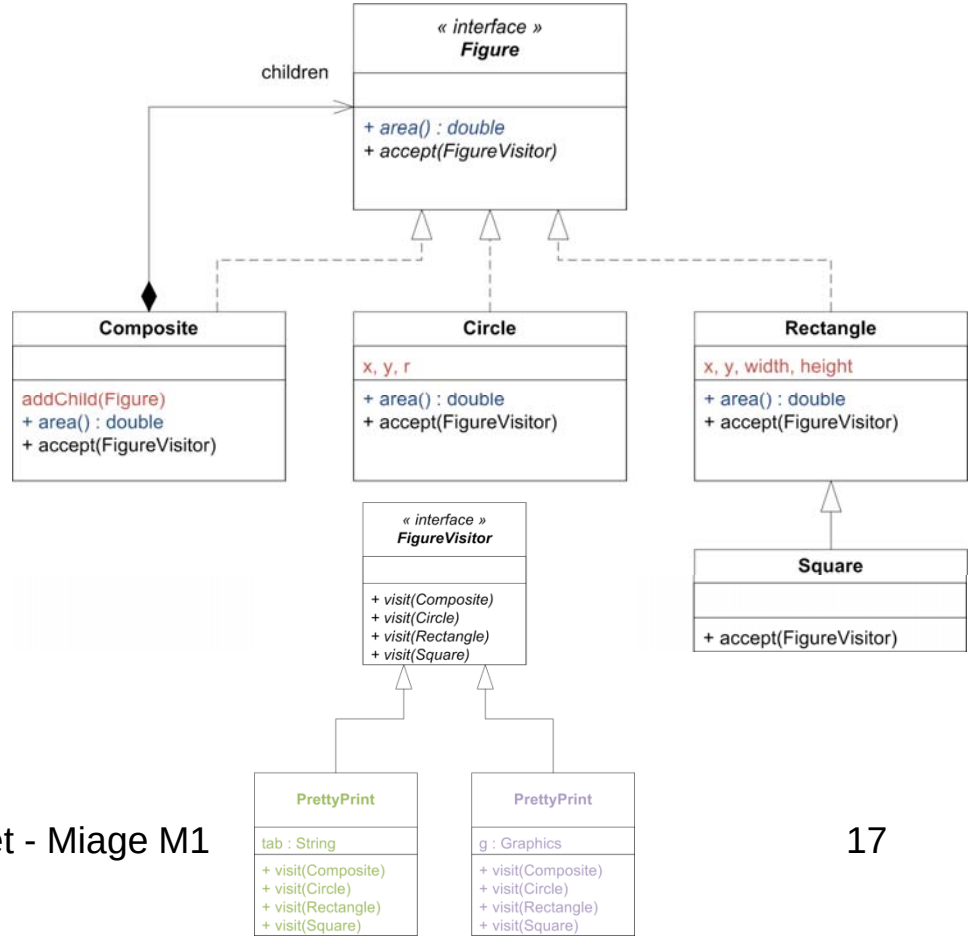
Métier Figure
Spécifique classe concrète
Service PrettyPrint
Service draw

Comparaison

Sans Visiteurs



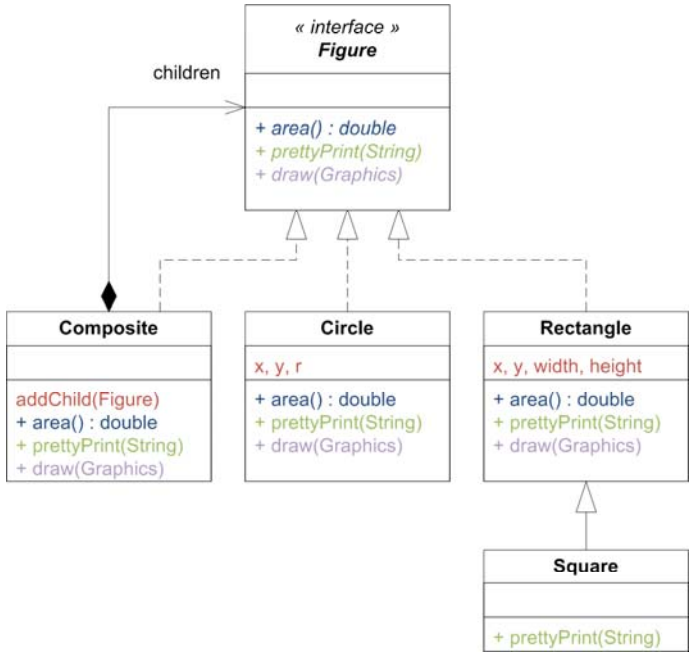
Avec Visiteurs



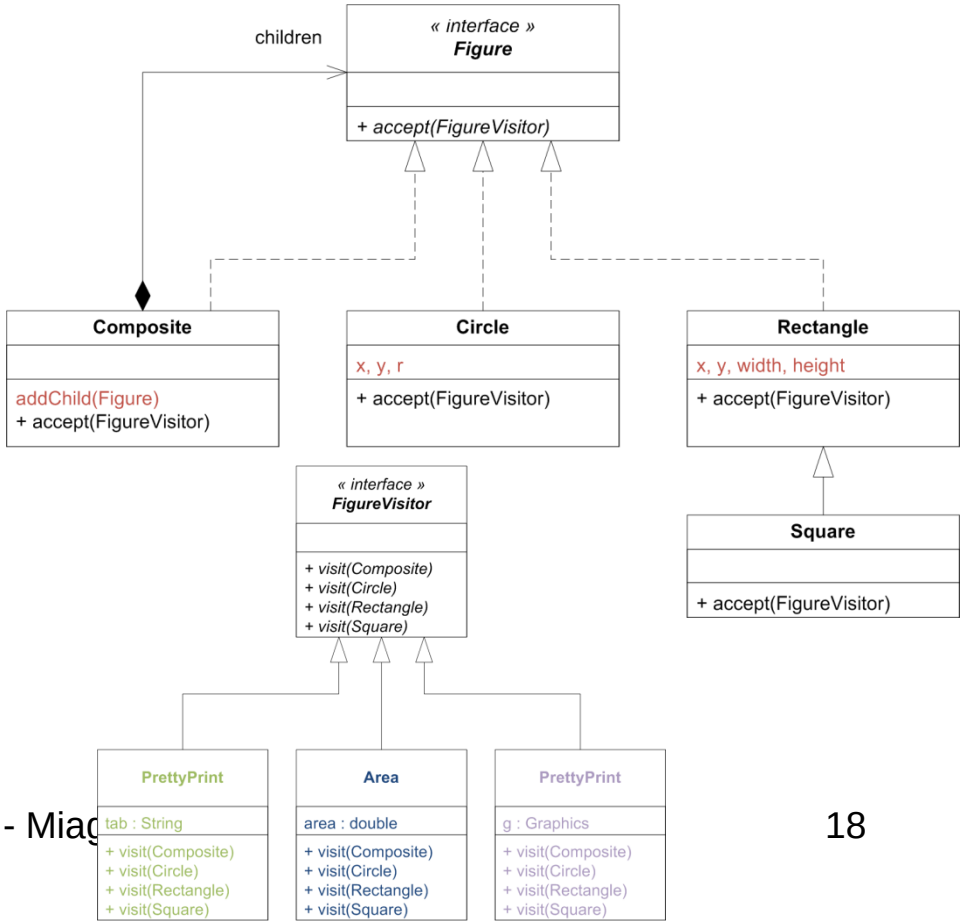
Service area
Spécifique classe concrète
Service PrettyPrint
Service draw

Comparaison

Sans Visiteurs



Avec Visiteurs



Université Nice Sophia Antipolis

Conclusions

- Conséquences
  - Ajout de nouvelles opérations très facile
  - Groupement/séparation des opérations communes
  - Ajout de nouveaux ConcreteElement complexe (opération dépendant du type…)
  - Visitor traverse des structures où les éléments sont de types complètement différents / Iterator
  - Accumulation d'état dans le visiteur plutôt que dans des arguments
  - Suppose que l'interface de ConcreteElement est assez riche pour que le visiteur fasse son travail => cela force à montrer l'état interne et à casser l'encapsulation