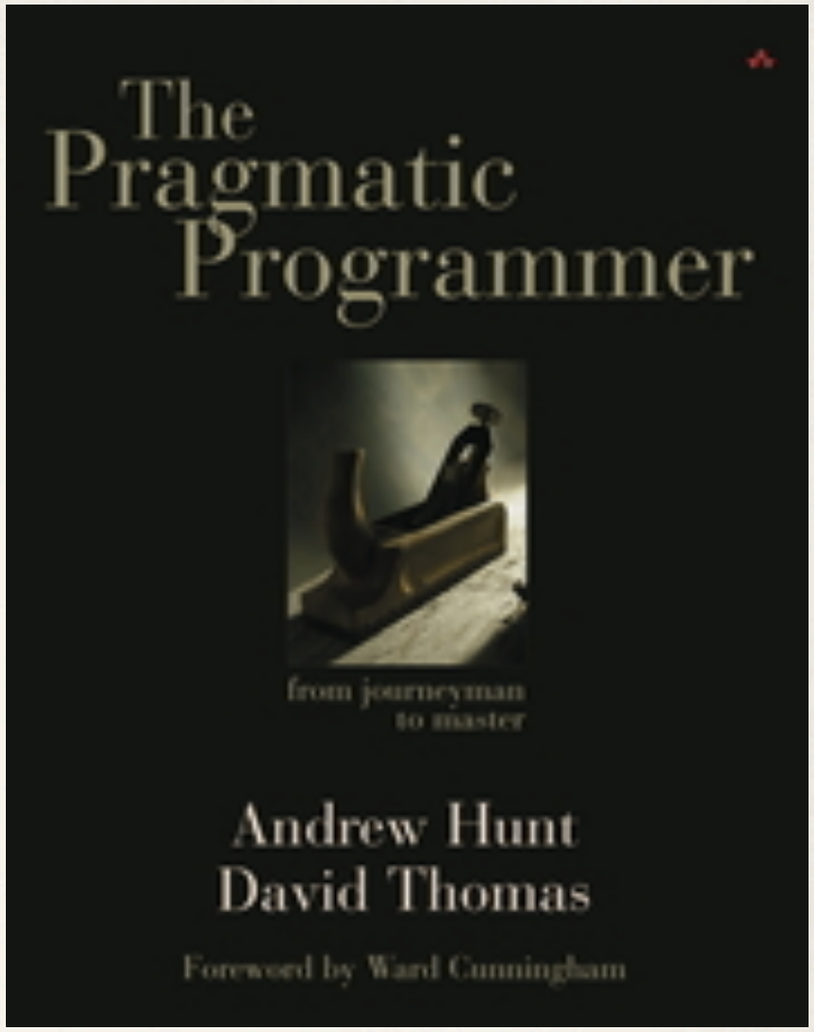
En guise d'introduction



Pragmatic Programming

The Pragmatic Programmer: From Journeyman to Master

by Andrew Hunt and David Thomas

Et les **principes SOLID**

Mireille Blay-Fornarino, Université Nice Sophia Antipolis, Département Info IUT, Septembre 2014

Objectifs de ce cours

- **Mieux comprendre votre rôle en tant que «Développeur»**
- Positionnement des différents cours qui auront lieu par la suite.
- Possibilité que certains cours ou éléments de cours soient donnés par les étudiants : Signalez vous !
- Ouverture à d'autres aspects de la conception et programmation avancée très probable en fonction des besoins.

  «Les développeurs avancés voient très vite l'intérêt, les débutants beaucoup moins. Quelques années plus tard, ils comprennent pourquoi c'était important!» Anonyme.

Au delà des méthodes

- Having a process is not the same as having the skills to carry out that process

— Jim Highsmith

Attention, version édulcorée (pragmatique?) du livre pour ne garder que ce qui peut vous «parler» dès à présent.

Les Hommes font la différence, donc VOUS...

Without excellent personnel, even good to excellent
Processes can only achieve marginal results.

— Capers Jones,
"Software Assessments, Benchmarks, and Best Practices", 2000

Pragmatic Attitude

Stone Soup and Boiled Frog

Image issue de http://www.inprogressweb.com/how-to-take-control-of-your-future-and-not-become-a-boiled-frog/

Be a Catalyst for Change

AND

Remember the Big Picture

La théorie des fenêtres cassées ou Eviter l'entropie du système

Etude urbaine en 1982 : http://en.wikipedia.org/wiki/Broken_windows_theory

Ne pas laisser de fenêtre cassée :

- Réparer les codes, Corriger les design dès que les défauts sont détectés.
- Si vous ne pouvez pas régler le problème, le circonscrire : annoter le code, noter «Not Implemented», ...
- Mais ne laisser pas des codes se déteriorer ou c'est l'ensemble de l'application qui en pâtira.

Prendre 10 s pour s'interroger

- Pourquoi je fais cela? Quel est mon objectif?
- Est-ce ainsi que cela doit être fait?
- Est-ce nécessaire de le faire?
- Que signifie «terminé» dans ce cas?

http://tck.io/posts/sharing_models.html

Ecrire du bon code : Don’t Repeat Yourself (DRY)

> Every piece of knowledge must have a single, unambiguous, authoritative representation within a system

Causes de duplication des codes :

- Imposition a priori de l’environnement,
- Inattention,
- Facilité,
- Multiplicité des développeurs

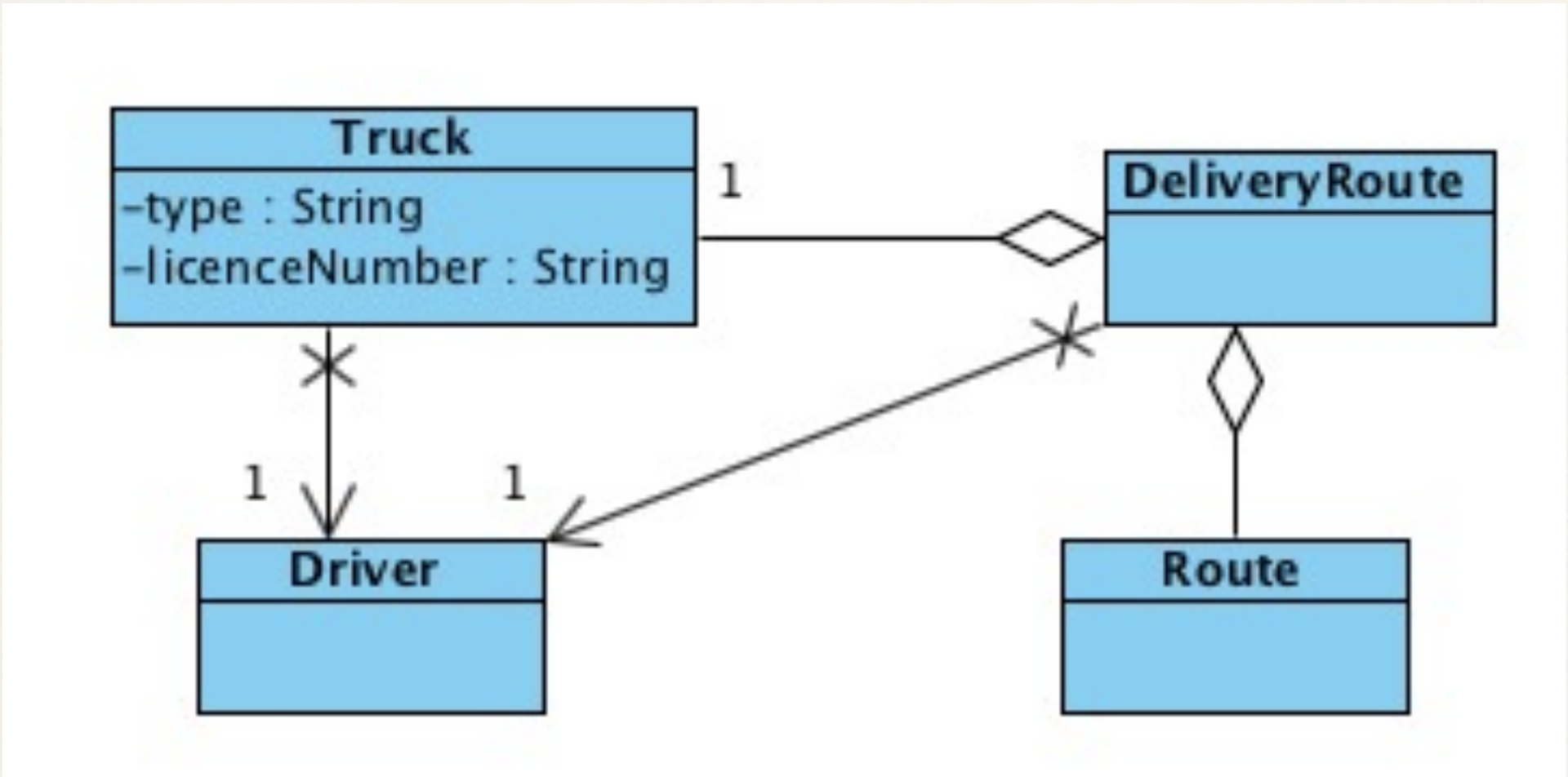
DRY (1) : Des exemples de duplications imposées et des solutions

- Documentation du code => ne garder les commentaires détaillés que pour le haut niveau.

- Multiples représentations d'une information (coté client et serveur par exemple, une classe miroir d'une table dans la BD) => des filtres, des générateurs de code, metadata et générateur, génération de la classe à partir de la BDD ou du schéma, ou du modèle,...

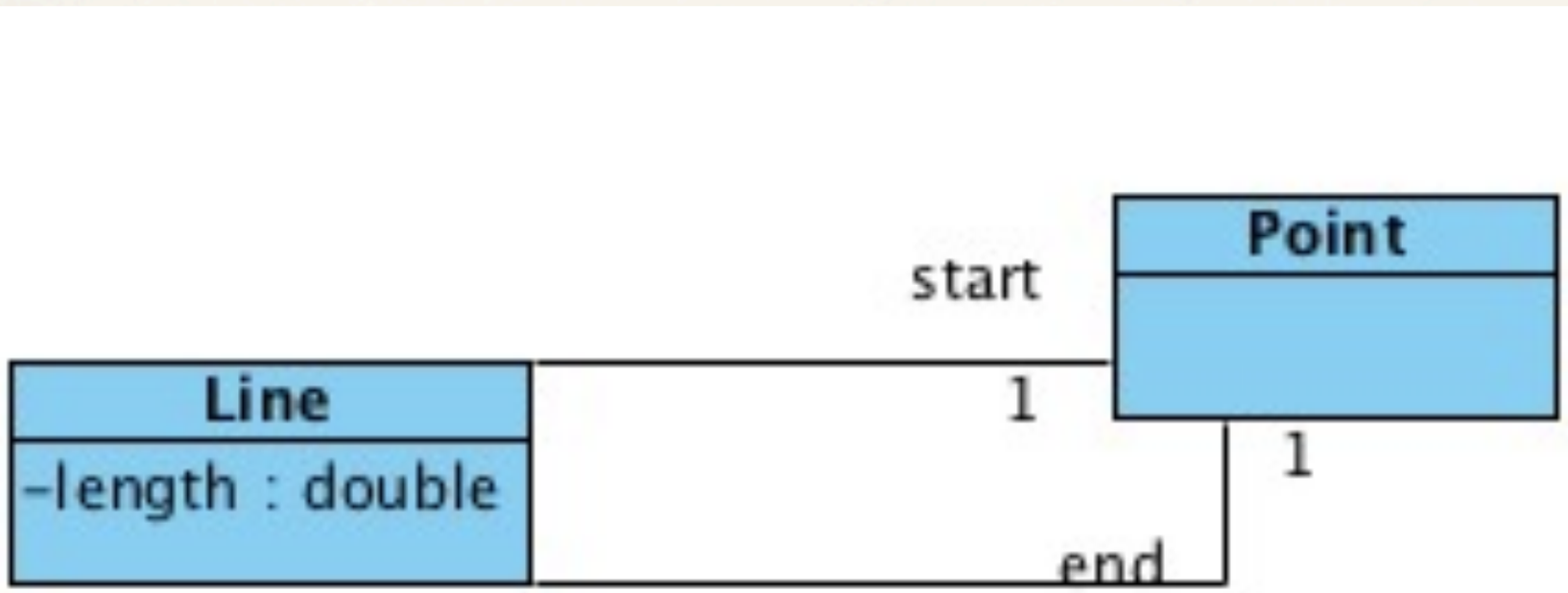
- Les langages forcent des duplications : utiliser des outils!

Dry (2) : Des exemples de duplication par inattention et des solutions



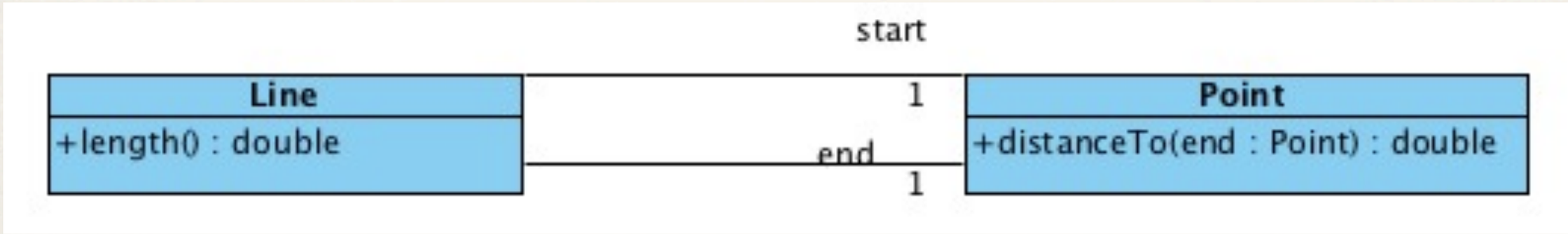
- Que faut-il modifier pour changer un chauffeur ? N'y a t'il pas une connaissance dupliquée?

Dry (2) : Des exemples de duplications par inattention et des solutions



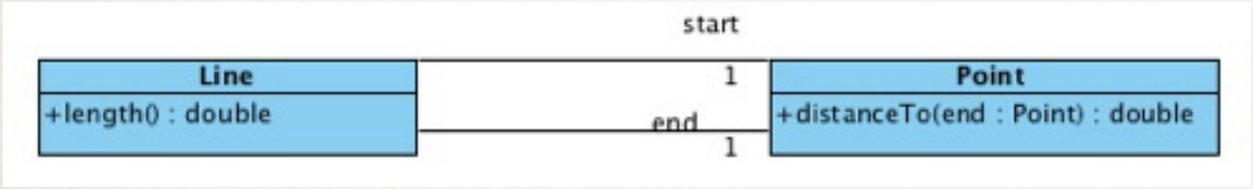
- Qu'est-ce qui est dupliqué?

Dry (2) : Des exemples de duplications par inattention et des solutions



- return start.distanceTo(end);

Dry (2) : Des exemples de duplications par inattention et des solutions



- **Optimisation** : le modèle ne change pas forcément

Version «paresseuse» : on ne calcule la longueur que si besoin !

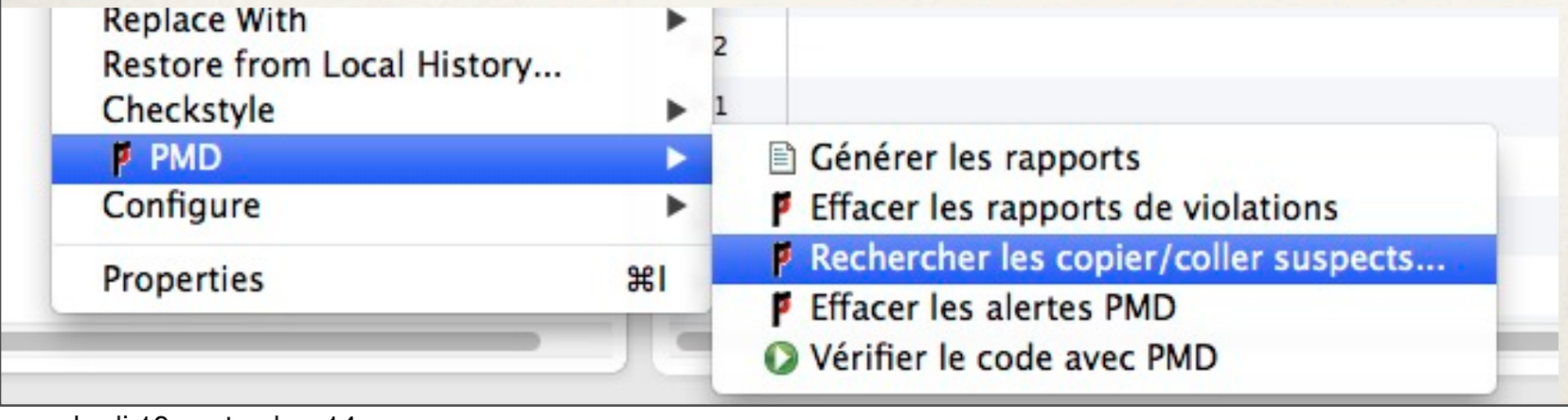
```
public class Line {

    private boolean changed;
    private double length;
    private Point start, end;

    public Line(Point start, Point end) {
        super();
        this.start = start;
        this.end = end;
        changed=true;
        this.getLength();
    }

    public Point getStart() {
        return start;
    }

    public void setStart(Point start) {
        this.start = start;
        changed = true;
    }
    public Point getEnd() {
        return end;
    }
    public void setEnd(Point end) {
        this.end = end;
        changed = true;
    }
    public double getLength() {
        if (changed) {
            length = start.distanceTo(end);
            changed = false;
        }
        return length;
    }

}
```

Dry(3) : Des exemples de duplications par multi-développeurs et des solutions

- Vérification du numéro de sécurité social ... 10 000 programmes définissant des vérification équivalentes

✓ Ok, un bon manager peut éviter certaines duplications

- Mais aussi la communication entre développeurs
- Les outils de recherche de codes dupliqués.



voir en TD.

Ecrire du bon code :
Faible couplage et la loi de Demeter

"Each unit should only talk to its friends."
"Don't talk to strangers."

Good fences make good neighbors.
Robert Frost, «Mending Wall»

Découplage et la loi de Demeter

- Il ne s'agit pas de ne pas avoir d'interactions entre les modules mais d'en limiter le nombre!
- Déléguer «totalement» : Si vous donnez une tâche à un «objet», vous ne voulez pas qu'il vous renvoie un autre objet à qui demander de faire le travail !

La méthode d'un objet ne doit appeler que des méthodes appartenant à:

- Lui-même
- Tous les paramètres passés dans
  - les objets qu'elle crée
  - les objets qui composent l'objet

Découplage et la loi de Demeter : Minimiser le couplage

```
foobar.thing.manager.xyzzy.doit(6)
```

Casser le couplage en donnant à chacun ses responsabilités

```
foobar.doit(6)

    thing.doit(6)

        manager.doit(6)

            xyzzy.doit(6)
```

Découplage et la loi de Demeter : Minimiser le couplage, exemples

```
public void showBalance(BankAccount acct) {
Money amt = acct.getBalance();
printToScreen(amt.printFormat());
}
```

«amt ne fait pas partie de l'objet et n'est pas passé en argument, on ne doit pas lui parler!»

```
public void showBalance(BankAccount acct) {
    acct.printBalance();
}
```

Découplage et la loi de Demeter : Minimiser le couplage, exemples

```
public class Colada {
   private Blender myBlender;
   private Vector myStuff;
   public Colada() {
    myBlender = new Blender();
    myStuff = new Vector();
   }

   private void doSomething() {
    myBlender.addIngredients(myStuff.elements());
   }
}
```

OK !

Découplage et la loi de Demeter : Minimiser le couplage, exemples

```
void processTransaction(BankAccount acct, int) {
    Person *who;
    Money amt = new Money();
    amt.setValue(123.45);
    acct.setBalance(amt);
    who = acct.getOwner();
    markWorkflow(who->name(), SET_BALANCE);
}
```

```
void processTransaction(BankAccount acct, int) {
    Money amt = new Money();
    amt.setValue(123.45);
    acct.setBalance(amt);
    markWorkflow(acct.name(), SET_BALANCE);
}
```

Quand peut-on violer la loi de Demeter ?

- Dans des cas où une optimisation est nécessaire
  - Des contraintes de vitesse ou de mémoire par ex.
- Si le code accédé est vraiment très stable «Black Box»
  - Aucun changement de l'interface peut raisonnable être attendu
- SINON NE VIOLEZ PAS CETTE LOI !!
  - A long terme les coûts peuvent être élevés.

Ecrire du bon code : Préférer la composition à l'héritage

> Inheritance is a powerful way to achieve code reuse, but it is not always the best tool for the job. Used inappropriately, it leads to fragile software. It is safe to use inheritance within a package, where the subclass and the superclass implementations are under the control of the same programmers. It is also safe to use inheritance when extending classes specifically designed and documented for extension. Inheriting from ordinary concrete classes across package boundaries, however, is dangerous.
>
> Joshua Bloch

http://verraes.net/2014/05/final-classes-in-php/

Héritage & Composition

- Préférer la composition à l'héritage
  - L'héritage a été mis en avant pour la réutilisation
    - Trop !
  - Souvent, l'héritage est rigide et la composition est souple

Composition

- Method of reuse in which new functionality is obtained by creating an object *composed of* other objects
- The new functionality is obtained by delegating functionality to one of the objects being composed
- Sometimes called *aggregation* or *containment*, although some authors give special meanings to these terms

Inheritance vs Composition Example

- This example comes from the book *Effective Java* by Joshua Bloch
- Suppose we want a variant of HashSet that keeps track of the number of attempted insertions. So we subclass HashSet as follows:

```
public class InstrumentedHashSet extends HashSet {

    // The number of attempted element insertions
    private int addCount = 0;

    public InstrumentedHashSet() {super();}
    public InstrumentedHashSet(Collection c) {super(c);}
    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
    }
```

http://uet.vnu.edu.vn/~chauttm/e-books/java/Effective.Java.2nd.Edition.May.2008.3000th.Release.pdf

Inheritance vs Composition Example (Continued)

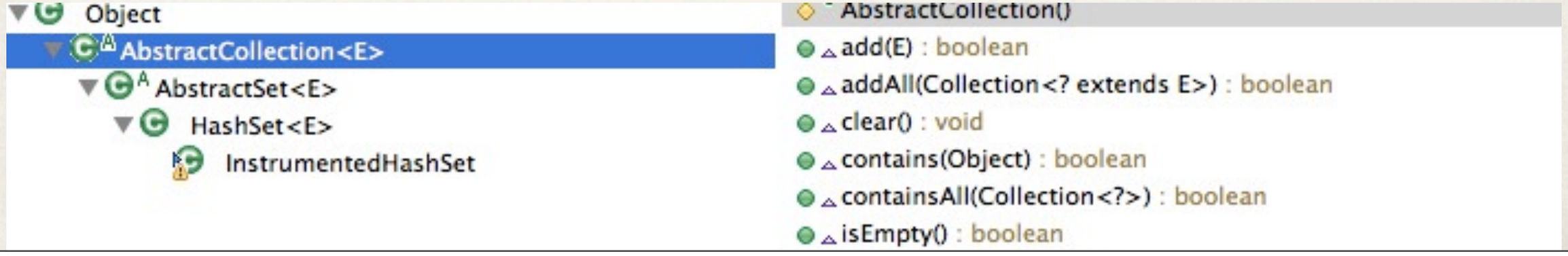
```
public boolean add(Object o) {
        addCount++;
        return super.add(o);
}

public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }
}
```

http://uet.vnu.edu.vn/~chauttm/e-books/java/Effective.Java.2nd.Edition.May.2008.3000th.Release.pdf

Inheritance vs Composition Example (Continued)

```
public static void main(String[] args) {
    InstrumentedHashSet s = new InstrumentedHashSet();
    s.addAll(Arrays.asList(new String[] {"Snap","Crackle","Pop"}));
    System.out.println(s.getAddCount());
}
```

- Let's test it! 6, why ??



Inheritance vs Composition Example (Continued)

```
public static void main(String[] args) {
    InstrumentedHashSet s = new InstrumentedHashSet();
    s.addAll(Arrays.asList(new String[] {"Snap","Crackle","Pop"}));
    System.out.println(s.getAddCount());
}
```

- Let's test it! 6, why ??

```
144:    public boolean addAll(Collection<? extends E> c)
145:    {
146:      Iterator<? extends E> itr = c.iterator();
147:      boolean modified = false;
148:      int pos = c.size();
149:      while (--pos >= 0)
150:        modified |= add(itr.next());
151:      return modified;
152:    }
```

Inheritance vs Composition Example (Continued)

- Implementation details of our superclass affected the operation of our subclass.
- The best way to fix this is to use composition. Let's write an InstrumentedSet class that is composed of a Set object. Our InstrumentedSet class will duplicate the Set interface, but all Set operations will actually be forwarded to the contained Set object.
- InstrumentedSet is known as a **wrapper** class, since it wraps an instance of a Set object.

http://uet.vnu.edu.vn/~chauttm/e-books/java/Effective.Java.2nd.Edition.May.2008.3000th.Release.pdf

Inheritance vs Composition Example (Continued)

```
public class InstrumentedSet implements Set {
    private final Set s;
    private int addCount = 0;

    public InstrumentedSet(Set s) {this.s = s;}

    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }

    public boolean addAll(Collection c) {
        addCount += c.size();
        return s.addAll(c);
    }

    public int getAddCount() {return addCount;}
```

```
// Forwarding methods (the rest of the Set interface methods)
public void clear() { s.clear(); }
public boolean contains(Object o) { return s.contains(o); }
public boolean isEmpty() { return s.isEmpty(); }
public int size() { return s.size(); }
public Iterator iterator() { return s.iterator(); }
public boolean remove(Object o) { return s.remove(o); }
public boolean containsAll(Collection c)
    { return s.containsAll(c); }
public boolean removeAll(Collection c)
    { return s.removeAll(c); }
public boolean retainAll(Collection c)
    { return s.retainAll(c); }
public Object[] toArray() { return s.toArray(); }
public Object[] toArray(Object[] a) { return s.toArray(a); }
public boolean equals(Object o) { return s.equals(o); }
public int hashCode() { return s.hashCode(); }
public String toString() { return s.toString(); }
}
```

Inheritance vs Composition Example (Continued)

```
public static void main(String[] args) {
    InstrumentedSet s1 = new InstrumentedSet(new HashSet());
    s1.addAll(Arrays.asList(new String[] {"Snap","Crackle","Pop"}));
    System.out.println(s1 + " " + s1.getAddCount());
}
```

```
[Pop, Snap, Crackle] 3
```

Inheritance vs Composition Example (Continued)

```
public static void main(String[] args) {
    List list = new ArrayList();
    InstrumentedSet s1 = new InstrumentedSet(new TreeSet(list));
    s1.addAll(Arrays.asList(new String[] {"Snap","Crackle","Pop"}));
    System.out.println(s1 + " " + s1.getAddCount());
}
```

```
[Crackle, Pop, Snap] 3
```

Inheritance vs Composition Example (Continued)

- Note several things:
  - This class is a Set
  - It has one constructor whose argument is a Set
  - The contained Set object can be an object of any class that implements the Set interface (and not just a HashSet)
  - This class is very flexible and can wrap any preexisting Set object
- Example:

```
List list = new ArrayList();
Set s1 = new InstrumentedSet(new TreeSet(list));

int capacity = 7;
float loadFactor = .66f;
Set s2 = new InstrumentedSet(new HashSet(capacity, loadFactor));
```

Advantages/Disadvantages of Inheritance

- Advantages:
  - New implementation is easy, since most of it is inherited
  - Easy to modify or extend the implementation being reused

- Disadvantages:
  - Breaks encapsulation, since it exposes a subclass to implementation details of its superclass
  - "White-box" reuse, since internal details of superclasses are often visible to subclasses
  - Subclasses may have to be changed if the implementation of the superclass changes
  - Implementations inherited from superclasses can not be changed at runtime

Advantages/Disadvantages Of Composition

- **Avantages**:
  - Objets contenus sont accessibles par la classe contenant uniquement à travers leurs interfaces
  - Réutilisation «Boîte noire» car les détails internes des objets contenus ne sont pas visibles
  - Bonne encapsulation
  - Réduit les dépendances de mise en œuvre
  - Chaque classe se concentre sur sa propre tâche
  - La composition peut être définie dynamiquement lors de l'exécution à travers des objets qui acquièrent des références à d'autres objets du même type
- **Inconvénients**:
  - Les systèmes résultant ont tendance à avoir plus d'objets
  - Les interfaces doivent être soigneusement définies afin d'utiliser de nombreux objets différents comme des blocs de composition

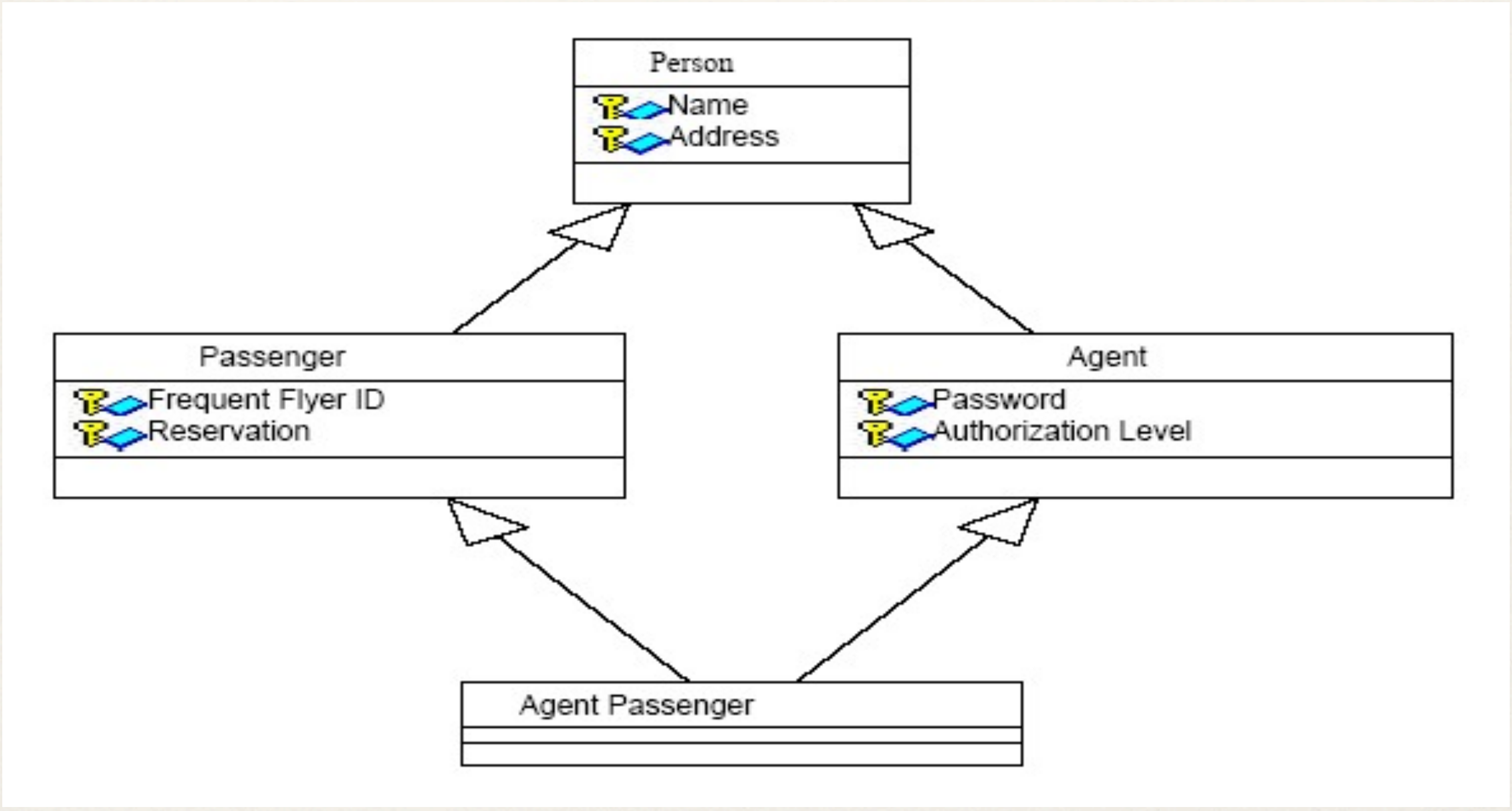
http://uet.vnu.edu.vn/~chauttm/e-books/java/Effective.Java.2nd.Edition.May.2008.3000th.Release.pdf

Coad's Rules of Using Inheritance

- Use inheritance only when all of the following criteria are satisfied:
  - A subclass expresses "is a special kind of" and not "is a role played by a"
  - An instance of a subclass never needs to become an object of another class
  - A subclass **extends**, rather than **overrides** or **nullifies**, the responsibilities of its superclass
  - A subclass does not extend the capabilities of what is merely an utility class

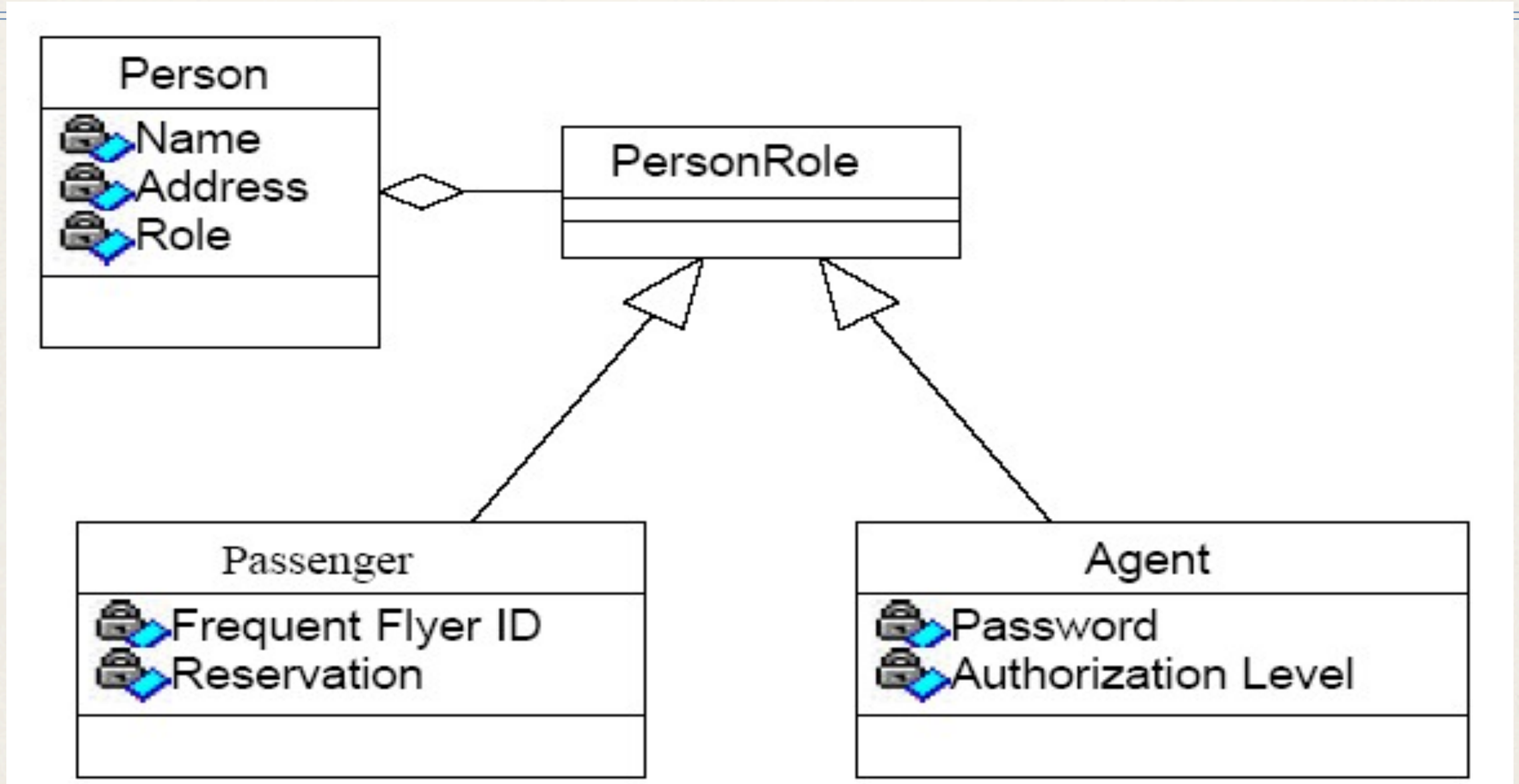
http://uet.vnu.edu.vn/~chauttm/e-books/java/Effective.Java.2nd.Edition.May.2008.3000th.Release.pdf

Inheritance ?
Example 1



http://uet.vnu.edu.vn/~chauttm/e-books/java/Effective.Java.2nd.Edition.May.2008.3000th.Release.pdf

Composition ? Example 2



http://uet.vnu.edu.vn/~chauttm/e-books/java/Effective.Java.2nd.Edition.May.2008.3000th.Release.pdf

Ecrire du bon code : Orthogonalité

Eliminate effects between unrelated Things

Gain en Productivité
Réduction du risque
Facilités pour la gestion des équipes
Simplification des tests

Ce n'est pas toujours possible, mais c'est alors un système complexe!

Ecrire du bon code : Orthogonalité, quelques éléments

- Prendre en compte l'orthogonalité dès la modélisation du système et jusque dans la mise en oeuvre : Architecture, Modèles, *Isoler les éléments qui sont extérieurs* pour minimiser les dépendances, ...

- Mais aussi dans la documentation en séparant le contenu et la forme!

C’est un des principaux arguments aux différents patrons étudiés.

Ecrire du bon code : Orthogonalité, exercices

Objectifs : découper des lignes en champs.

```
class Split1 {
      public Split1(InputStreamReader rdr) { ...
      public void readNextLine() throws IOException { ...
      public int numFields() { ...
      public String getField(int fieldNo) { ...
    }

class Split2 {
      public Split2(String line) { ...
      public int numFields()     { ...
      public String getField(int fieldNo) { ...
    }
```

Question : Quel design présente le plus d'orthogonalité ?

Ecrire du bon code : Orthogonalité, quelques éléments

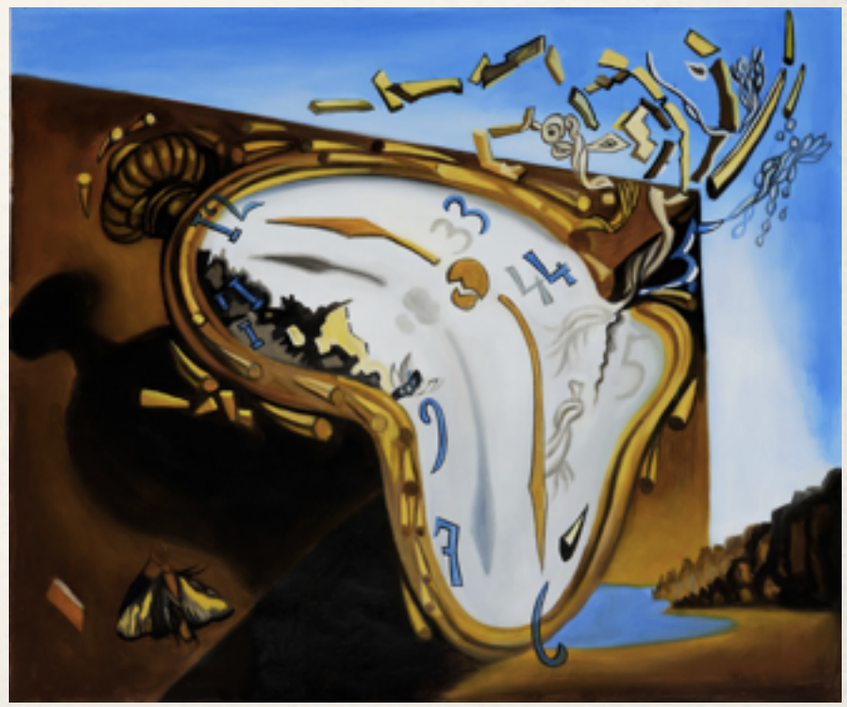
- Ecrire des codes «timides» : ils ne révèlent que ce qui est nécessaire et offrent peu de dépendances aux autres codes (Loi de Demeter) : pour modifier l'état d'un objet lui demander de le faire!
- Eviter les variables globales, et attention aux singletons.

Ecrire du bon code : Orthogonalité, quelques éléments

- Des techniques dédiées peuvent aussi être utilisées : annotation java pour la persistance, programmation par aspects, fichiers de propriétés (sécurité)

Ecrire du bon code :
Couplage temporel

«We need to allow for concurrency and to think about decoupling any time or order dependencies. In doing so, we can gain flexibility and reduce any time-based dependencies in many areas of development: workflow analysis, architecture, design, and deployment» «Hunt, Thomas»



Couplage temporel

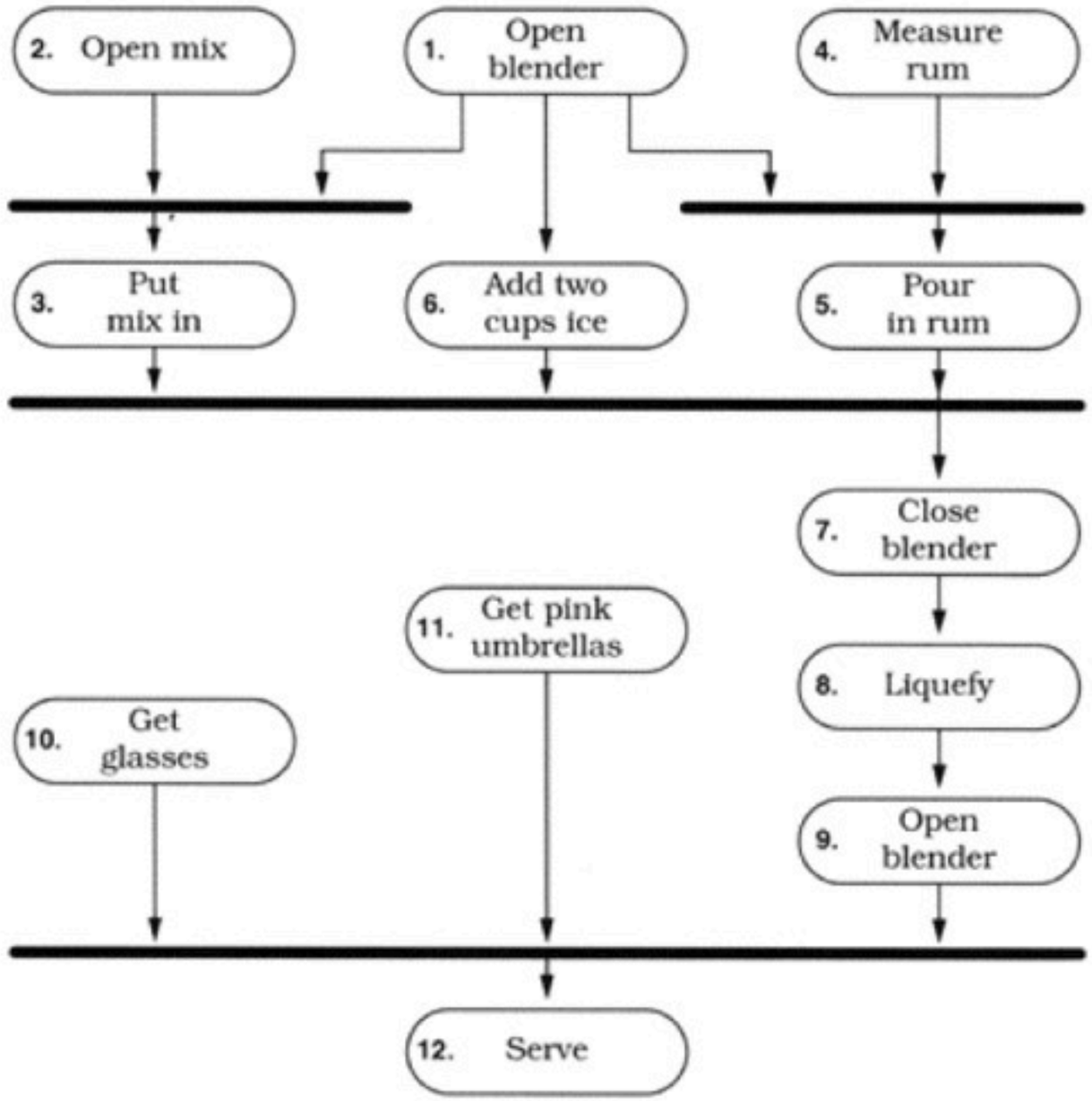
- Construire un diagramme d'activités et Identifier les activités qui peuvent se faire en parallèle

- Construire des architectures qui supportent le découplage

Couplage temporel, exemple

- Construire un diagramme d'activités et Identifier les activités qui peuvent se faire en parallèle

1. Open blender
2. Open piña colada mix
3. Put mix in blender
4. Measure 1/2 cup white rum
5. Pour in rum
6. Add 2 cups of ice
7. Close blender
8. Liquefy for 2 minutes
9. Open blender
10. Get glasses
11. Get pink umbrellas
12. Serve



cf. cours ultérieur sur les diagrammes d'activités

Cohésion temporelle

- Définition: Les éléments d'un composant sont liés par le «temps».
  - Difficile de modifier la structure car cela implique de vérifier le comportement de nombreux composants.
  - Le composant à peu de chance d'être réutilisable.
- **Exemple** : Une méthode d'initialisation du système contient la totalité du code d'initialisation de toutes les parties du système. Beaucoup d'activités différentes se déroulent pendant l'initialisation.
- **Amélioration** : La méthode d'initialisation du système envoie un message d'initialisation à chaque composant. Chaque composant s'initialise lui-même au moment de son instanciation..

Conclusion

- Nous venons de voir quelques principes pour un «bon» code.
- Fort de cette introduction et de l'attitude associée, nous nous intéressons à présent aux Principes SOLID
- Nous reviendrons sur «Pragmatic Programming» ensuite