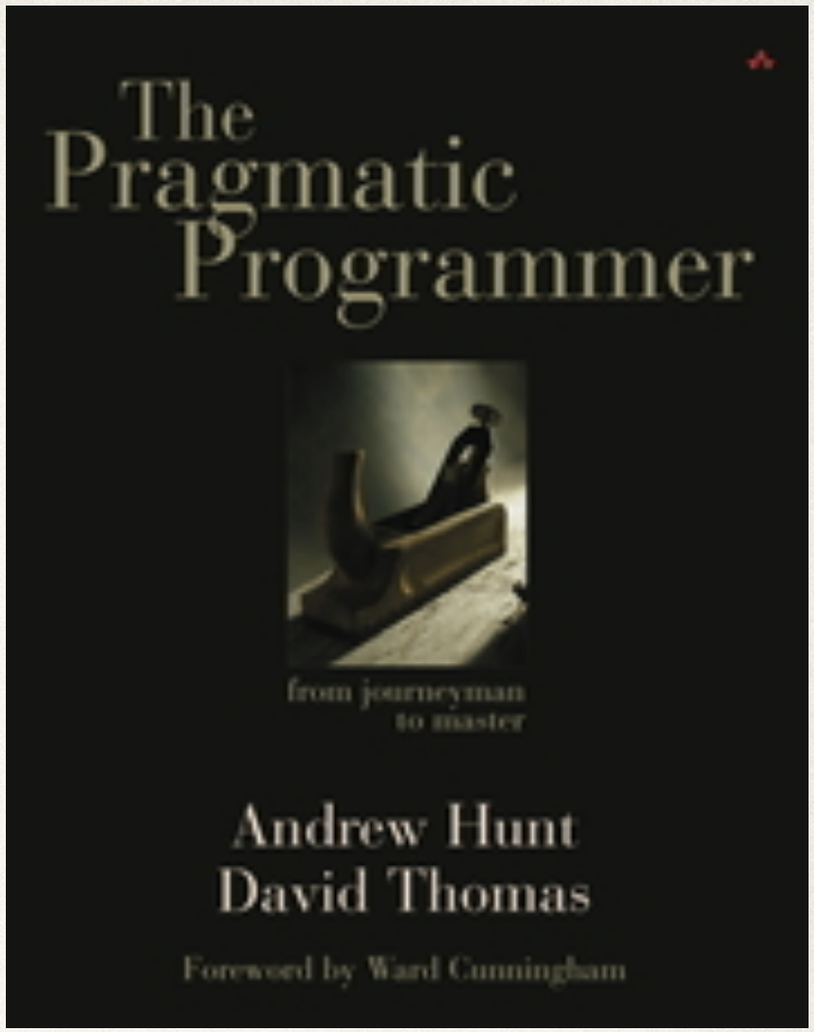



# Pragmatic Programming

**The Pragmatic Programmer: From Journeyman to Master**

by Andrew Hunt and David Thomas

---

*Mireille Blay-Fornarino, Université Nice Sophia Antipolis, Département Info Iut, Septembre 2018*

In other words, aggregation is meaningless. If you disagree, I challenge you to provide me with definitions of the semantics of aggregation and association, where the distinction between the two is unambiguous. **1/10/18**

*Https://Alessandrorossini.Org/Opinion-Why-Is-Model-Driven-Engineering-Unpopular-In-Industry-And-What-Can-We-Do-About-It/Date*

# Rappel : Concevoir une application, c'est imaginer une solution faîtes pour cette application : du sur-mesure! Et vous devez pouvoir expliquer pourquoi c'est la bonne solution !

Pour cela vous pouvez adapter du « prêt à concevoir »… mais l'explication est toujours nécessaire.

Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.
*Martin Fowler*

# L'art du «Codage»

«Conventional wisdom says that once a project is in the coding phase, the work is mostly mechanical, transcribing the design into executable statements. We think that this attitude is the single biggest reason that many programs are ugly, inefficient, poorly structured, unmaintainable, and just plain wrong.
Coding is not mechanical. If it were, all the CASE tools that people pinned their hopes on in the early 1980s would have replaced programmers long ago. There are decisions to be made every minute—decisions that require careful thought and judgment if the resulting program is to enjoy a long, accurate, and productive life.»

Hunt, Thomas «The pragmatic Programmer»

1. Eviter la duplication

2. Composition versus Héritage

3. Optimisation

4. Estimation des algorithmes

5. Programmation par coïncidence

6. Point de vue sur le «refactoring»

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

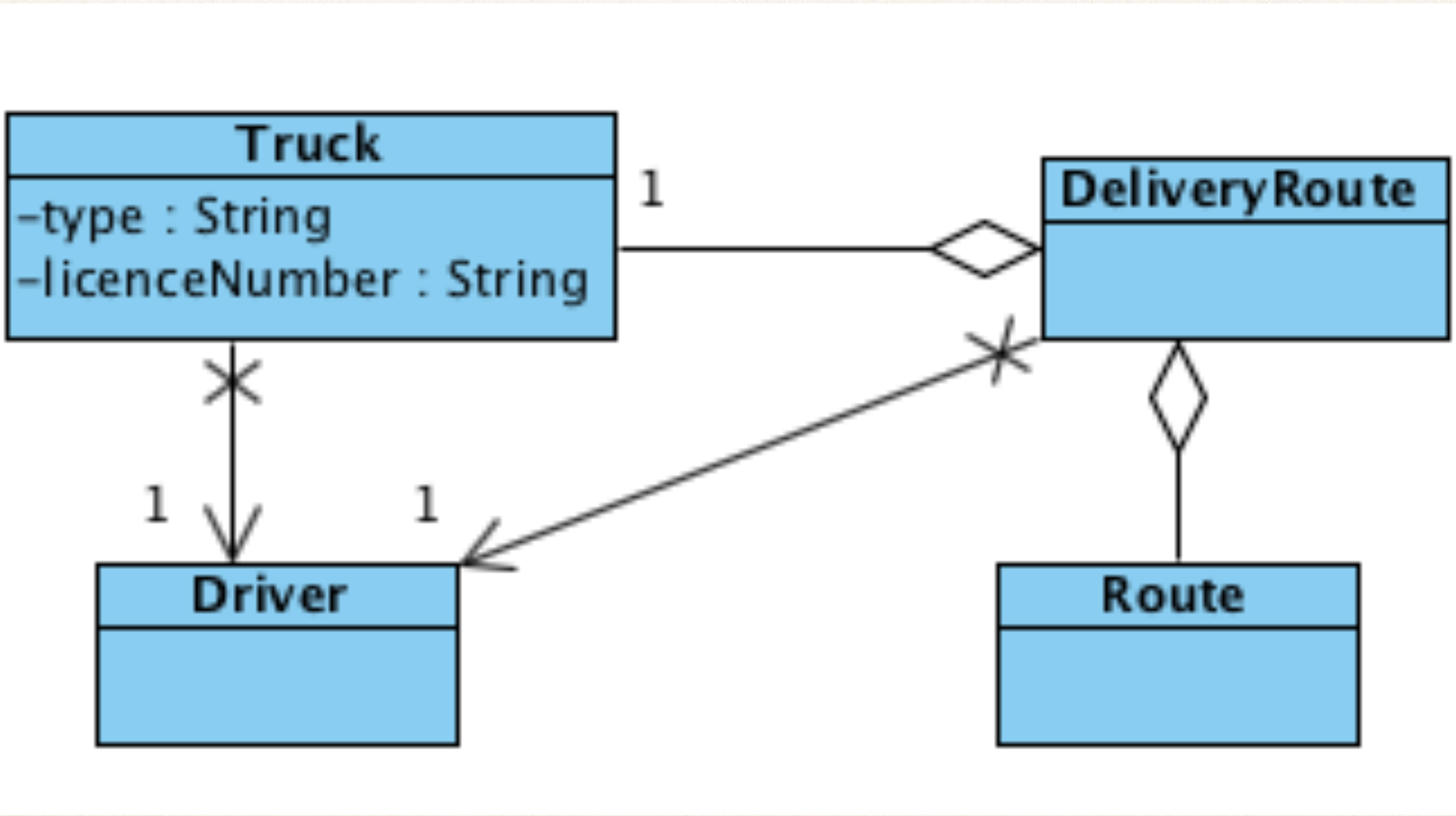

# DRY (1) : Des exemples de duplications imposées et des solutions

- Documentation du code
  - Les commentaires ne devraient pas ressembler à de la duplication de code

- Multiples représentations d'une information
  - (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,...

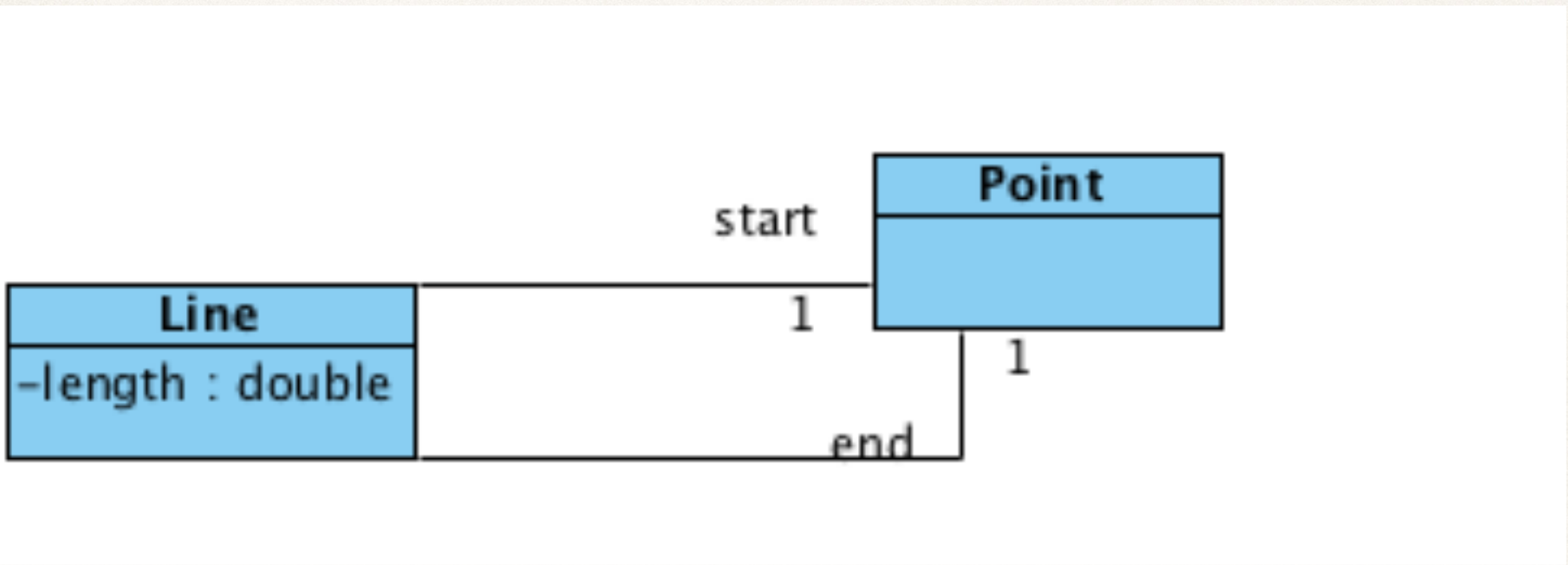
- Les langages/frameworks  forcent des duplications :  utiliser des outils!

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



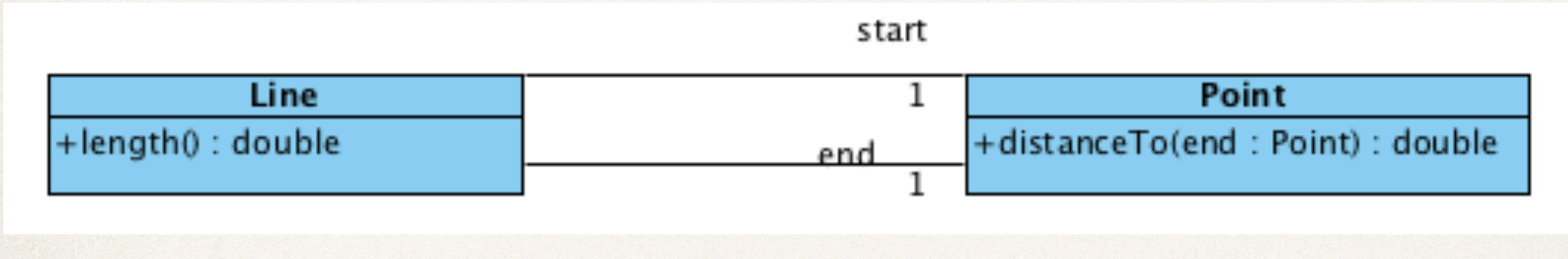

- Que faut-il modifier pour changer un chauffeur ? N'y a t'il pas une connaissance dupliquée?

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



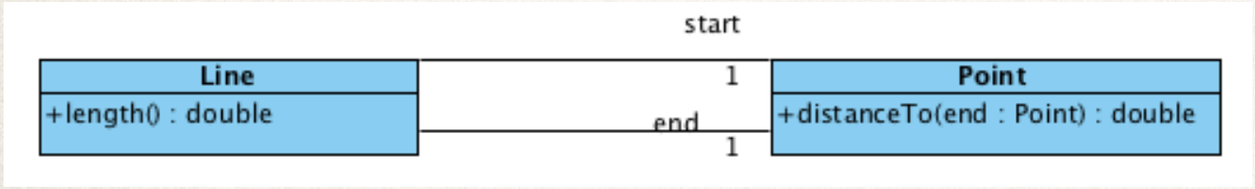

- Qu'est-ce qui est dupliqué?

# 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 !

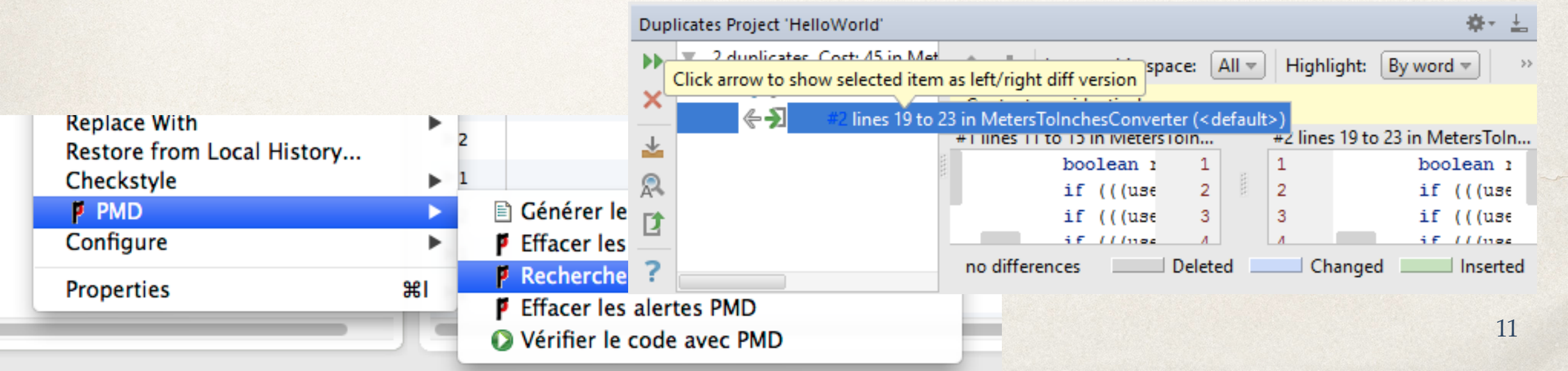
```
public class Line {

    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

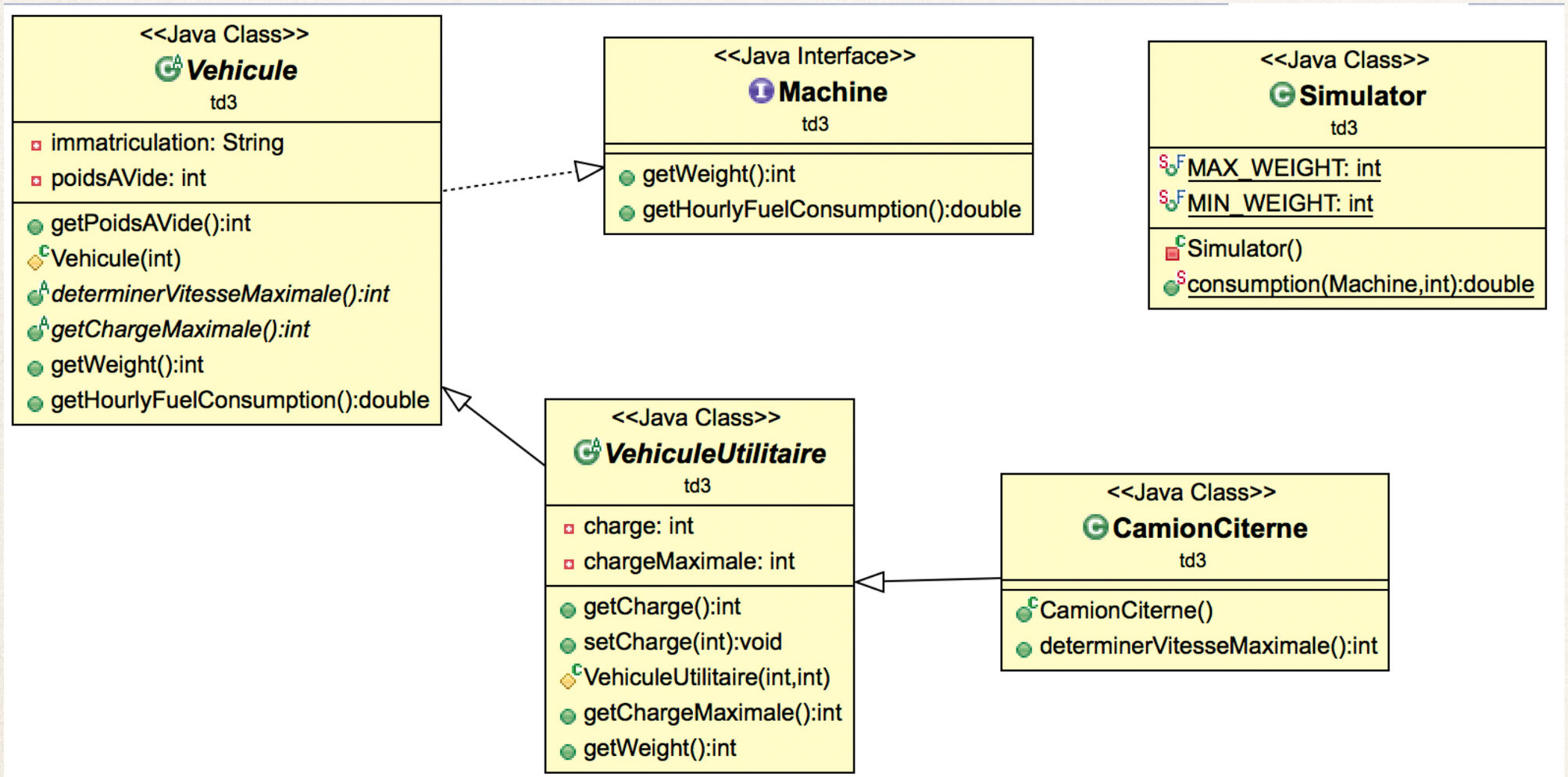
- Vérification du numéro de sécurité social ... 10 000 programmes définissant des vérification équivalentes

✓ Ok, un bon manager peut éviter certaines duplications

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



# Dry(4) : ré-utilisez vos codes pour ne pas dupliquer




La dépendance entre Simulator et Machine n'a pas été visualisée.

# Dry(4) : ré-utilisez vos codes pour ne pas dupliquer

```
public abstract class Vehicule implements Machine {
    private  String immatriculation = "";
    private int poidsAVide = 425;

     protected Vehicule(String immatriculation, int poidAVide){
        this.poidsAVide = poidAVide;
        this.immatriculation = immatriculation;
}

 protected Vehicule(int poidAVide){
     this("Not DEFINED",poidAVide);
}
```

# Dry(4) : ré-utilisez vos codes pour ne pas dupliquer

```
public abstract class Vehicule implements Machine {
private int poidsAVide = ..;
    protected Vehicule(int poidAVide){
        ...
```

```
public abstract class VehiculeUtilitaire extends Vehicule {
    protected VehiculeUtilitaire(int poidAVide, int chargeMaximale){
        super(poidAVide);
        this.chargeMaximale = chargeMaximale;
    }

                public class CamionBache extends VehiculeUtilitaire{
                    public CamionBache() {
                        super(4000, 20000);
                    }
```

Au niveau modèle, cela ne se voit pas, vraiment.

# Dry(4) : ré-utilisez vos codes pour ne pas dupliquer

**Dans la classe Contrôleur**

```
//Méthode utilitaire
    private Forum getForum() {
    String nomDuForum =
         ui.getNomDuForum(registre.getNomForums());
     if (!registre.exist(nomDuForum)){
          ui.afficher("Ce Forum n'existe pas");
          return null;
      }
      return registre.getForum(nomDuForum);
}
```

```
private void creerCanal(){
    Forum forum = getForum();

    if (forum != null) {
    String nomDeCanal = ui.getNomCanal(forum.getNomCanaux()
                ....
```

```
private void posterMessage() {
    Forum forum = getForum();
     .... }
```

```
private void lireMessage() {
    Forum forum = getForum();
```

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

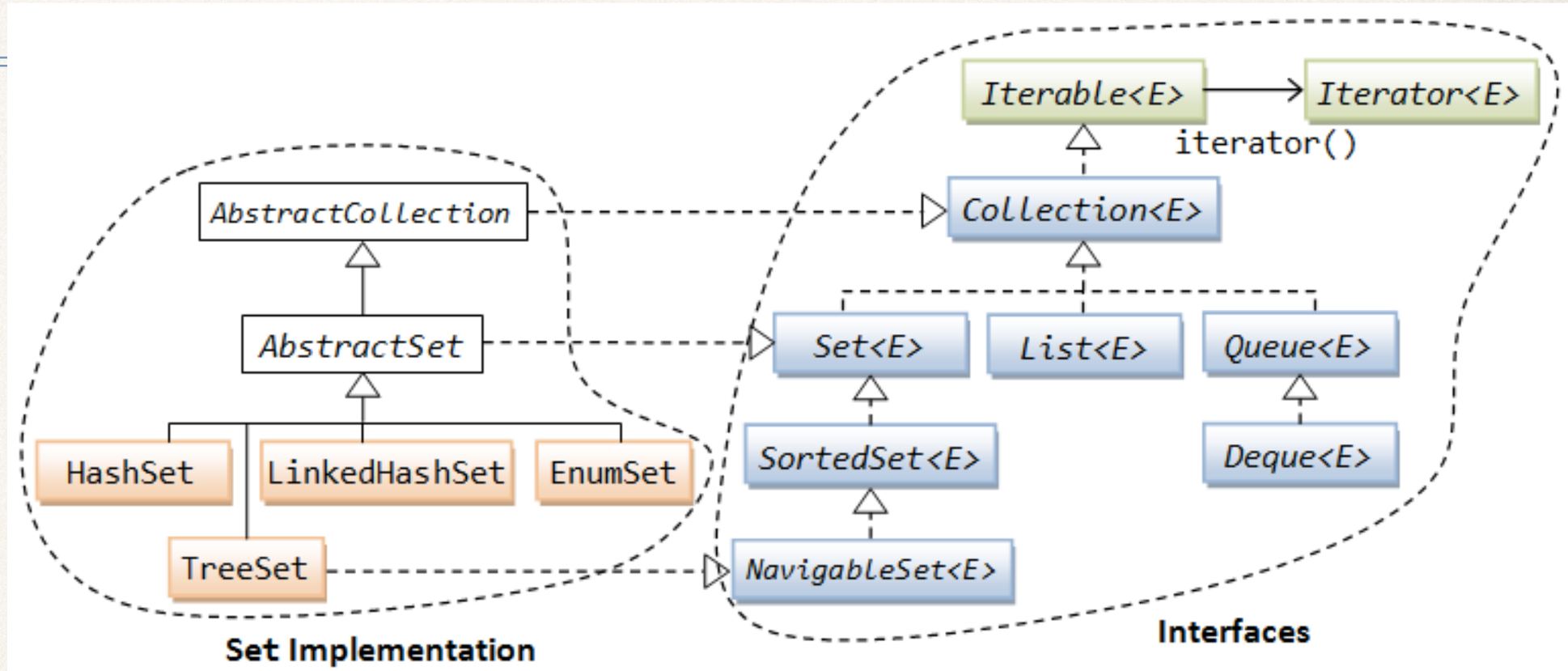
> Inheritance is a powerful way to achieve code reuse, but it is not always the best tool for the job. Used inappropriately, it leads to fragile software. It is safe to use inheritance within a package, where the subclass and the superclass implementations are under the control of the same programmers. 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

# Inheritance vs Composition Example




```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

This class implements the Set interface, backed by a **hash table** (actually a HashMap instance).
This class offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance (the number of buckets).
Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

# Inheritance vs Composition Example

- 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)

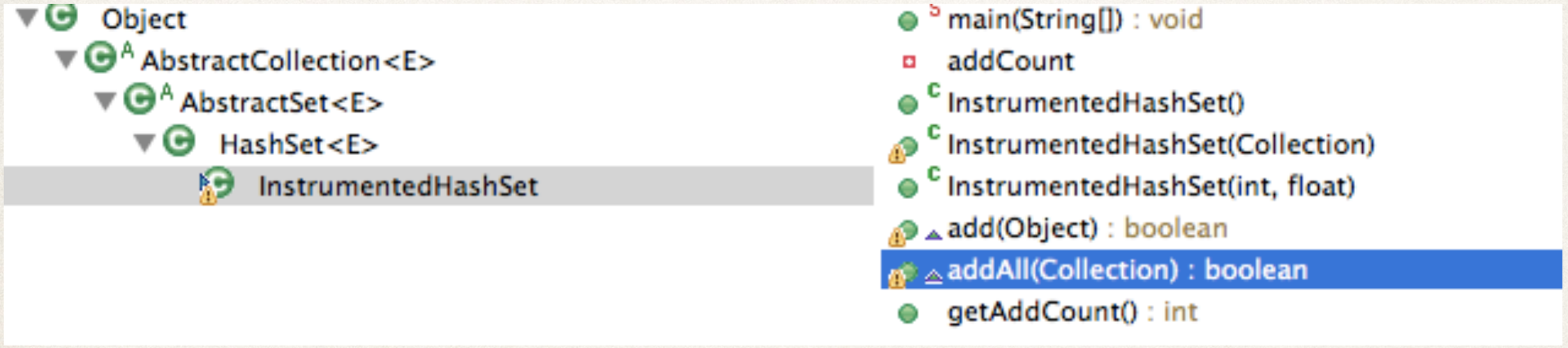
```
public boolean add(Object o) {
    addCount++;
    return super.add(o);
}

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

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

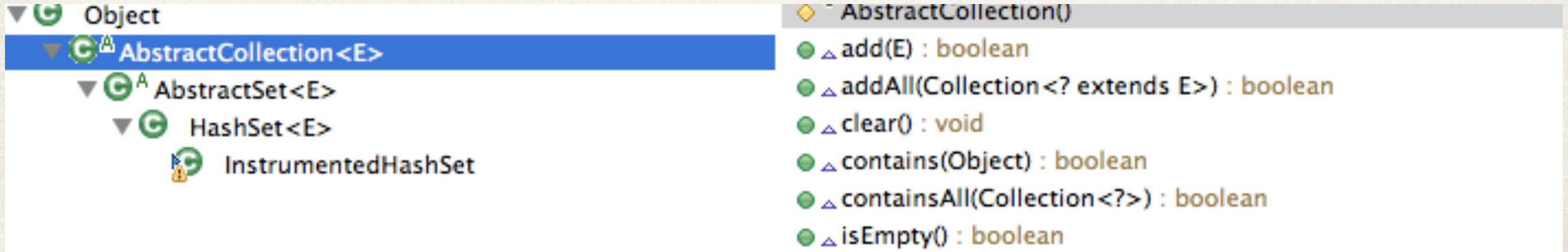
http://uet.vnu.edu.vn/~chauttm/e-books/java/Effective.Java.2nd.Edition.May.2008.3000th.Release.pdf

# 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)

- L'implementation des superclasses a affecté l'opération de la sous-classe.
- Une autre approche est d'utiliser la composition : La nouvelle classe «InstrumentedSet» n'est plus une sorte de «Set» mais est composée d'un ensemble d'objets.
- Cette classe « InstrumentedSet» duplique l'interface «Set», mais toutes les opérations sont déléguées à l'objet ensemble contenu.
- *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)

- Notez
  - Cette classe est  un « Set »
  - Elle a un constructeur qui prend en paramètre un « 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

- Avantages:
  - La nouvelle implémentation est facile, car la plupart des méthodes sont héritées
  - Facile à modifier ou à étendre
- Désavantages:
  - Rompt l'encapsulation, car elle expose une sous-classe aux détails d'implémentation de sa super-classe
  - Réutilisation en "boîte blanche", car les détails internes des superclasses sont souvent visibles par les sous-classes
  - Les sous-classes devront peut-être être modifiées si la mise en œuvre de la superclasse change
  - Les implémentations héritées des superclasses ne peuvent pas être modifiées à l'exécution

# 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



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

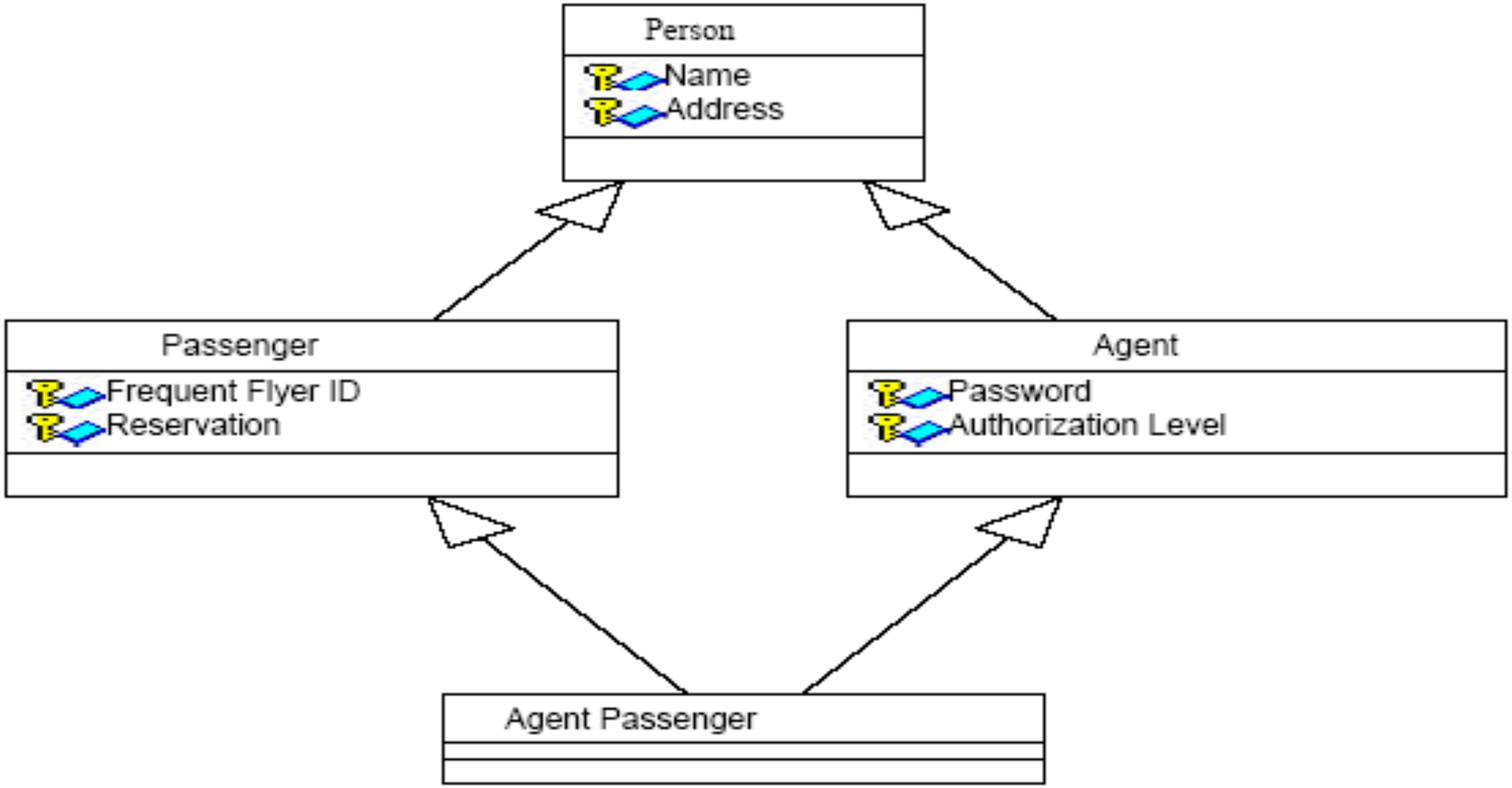
# Coad's Rules of Using Inheritance

- 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

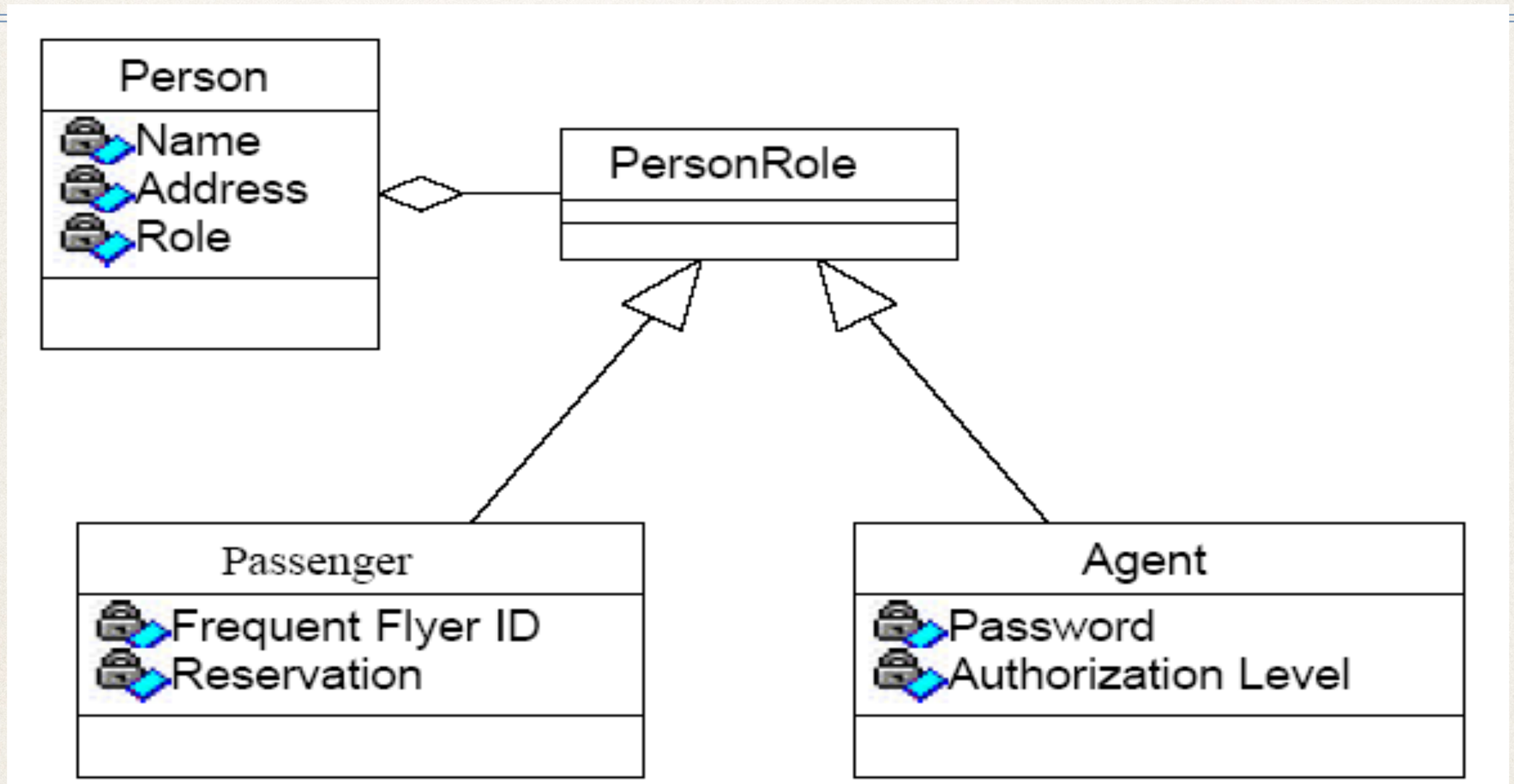
# Inheritance ?
# Example 1






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

# Premature Optimization

# Premature Optimization

```php
if (isset($frm['title_german'][strcspn($frm['title_german'], '<>')]))
{
    // ...
}
```

Optimiser les points vraiment utiles !
Ne mettez pas en péril la lisibilité et la maintenance de votre code pour de pseudo  micro-optimisations!
Ne gâcher pas votre temps!

http://nikic.github.io/2011/12/27/Dont-be-STUPID-GRASP-SOLID.html

# « Premature Optimization » ne signifie pas « faire n'importe quoi »

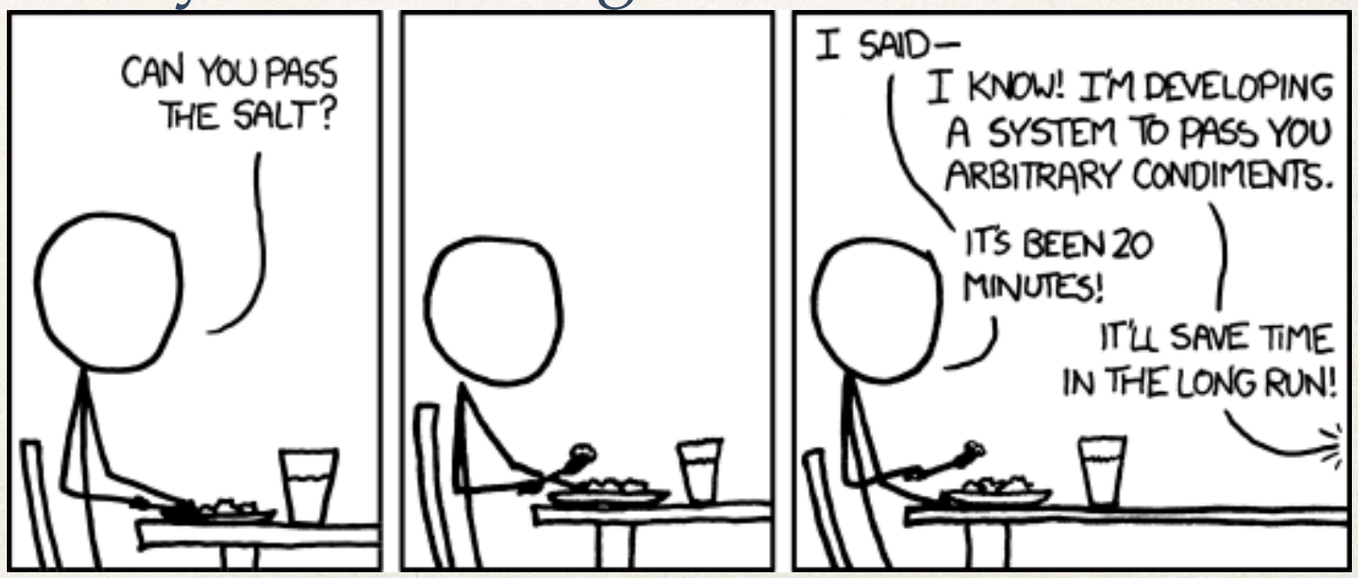
When I design a system, I do a rough estimate of how many times each piece of code will be executed per frame and use that to guide the design:

1-10 Performance doesn't matter. Do whatever you want.

100 Make sure it is O(n), data-oriented and cache friendly

1000 Make sure it is multithreaded

10000 Think really hard about what you are doing




http://altdevblog.com/2011/12/08/a-pragmatic-approach-to-performance/

# Estimation d'algorithme

# Penser à «Estimer» vos algorithmes

- Comment le programme se comportera s'il y a 1000 enregistrements? 1 000 000? Quelle partie optimiser?
- Quelles dépendances entre par exemple la taille des données (longueur d'une liste, par exemple) et le temps de calcul? et la mémoire nécessaire?

- S'il faut 1s pour traiter 100 éléments, pour en traiter 1000, en faut-il :
  - 1, (O(1)) : temps constant
  - 10 (O(n)) : temps linéaire
  - 3 (O(log(n)) : temps logarithmique
  - 100 ($O(n^2)$)
  - $10^{263}$ ($O(e^n)$) : temps exponentiel

# Penser à «Estimer» vos algorithmes

- Tester vos estimations

- Optimiser si cela est **utile** et en tenant compte du **contexte**.

# Eviter la Programmation par coïncidence

# Eviter la programmation par coïncidence

«Developers who don't actively think about their code are programming by coincidence—the code might work, but there's no particular reason why »

Hunt, Thomas «The pragmatic Programmer»

- «Mon code est tombé en marche, enfin !»...
  - coïncidence? Accidents d'implémentation ?

paint(g);
invalidate();
validate();
revalidate();
repaint();
paintImmediately(r);

```
public void reinit(){
    size(650, 550);
    background(255, 255, 255);
    image(loadImage("background.png"), 0, 0);
}

 public void setup() {
  frameRate(4);
  reinit();
  memoriseCarts() ;....
  // on veut garder la main sur le jeu c'est mousePressed qui stimule des re
  noLoop();
}

public void draw() {
 afficherJoueurs();
 //afficherCartes();
}
```

# Eviter la programmation par coïncidence

«Developers who don't actively think about their code are programming by coincidence—the code might work, but there's no particular reason why »

Hunt, Thomas «The pragmatic Programmer»

- Ben, chez moi, ça marche...
- coïncidence? Accidents de contexte ? Hypothèses implicites ?

- «A oui!  ça ne peut pas marcher parce que tu n'as pas mis le code sous bazarland»

# «Refactoring» ou l'art du «jardinage logiciel»

« Rather than construction, software is more like gardening—it is more organic than concrete. You plant many things in a garden according to an initial plan and conditions. Some thrive, others are destined to end up as compost. You may move plantings relative to each other to take advantage of the interplay of light and shadow, wind and rain. Overgrown plants get split or pruned, and colors that clash may get moved to more aesthetically pleasing locations. You pull weeds, and you fertilize plantings that are in need of some extra help. You constantly monitor the health of the garden, and make adjustments (to the soil, the plants, the layout) as needed»

Hunt, Thomas «The pragmatic Programmer»

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

Codez toujours en pensant que celui qui maintiendra votre code est un psychopathe qui connait votre adresse.

*Martin Golding*

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 détériorer ou c'est l'ensemble de l'application qui en pâtira.

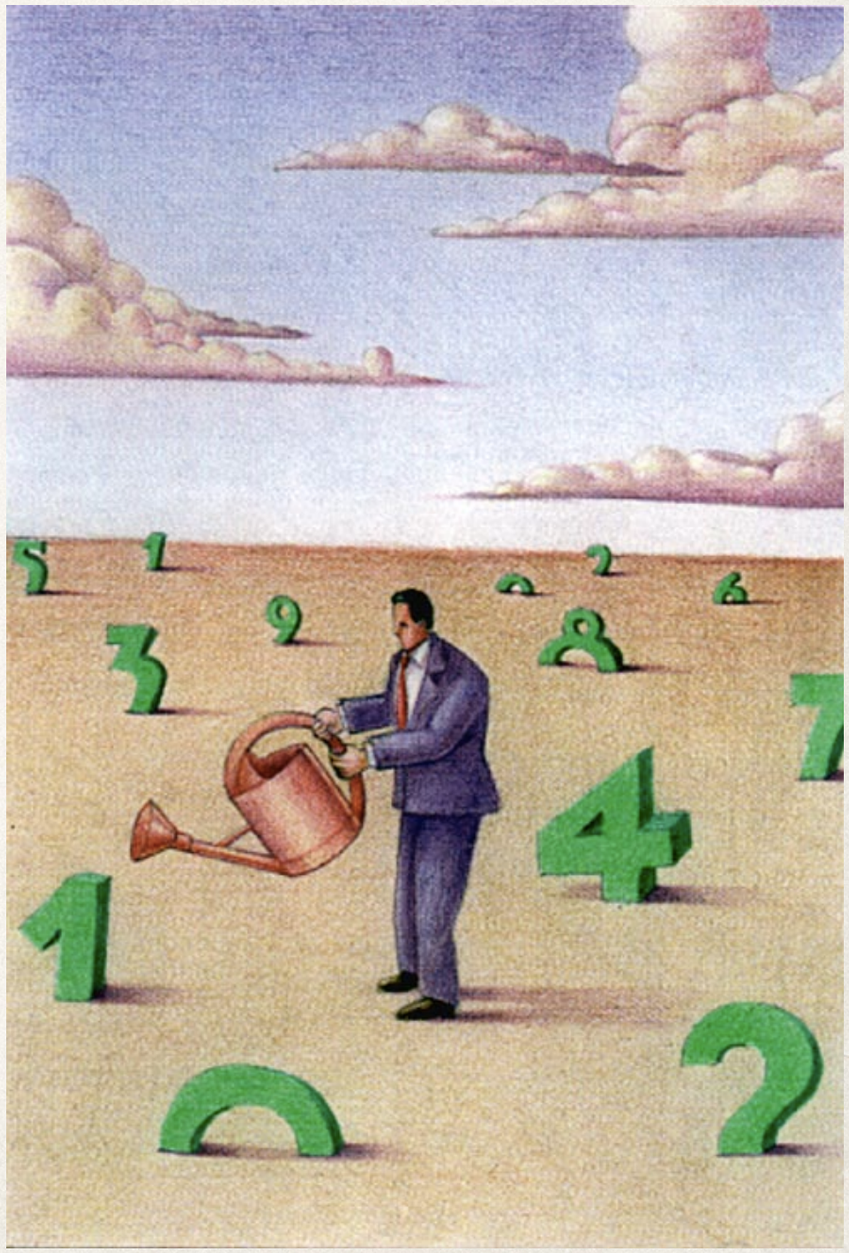
- Etude urbaine en 1982 : http://en.wikipedia.org/wiki/Broken_windows_theory

# «Refactoring» : quand ?

- Pour éliminer les «fenêtres cassées»
- Pour améliorer le design : duplication (DRY), couplage, performance, ...
- Pour ajuster en fonction des besoins et des demandes de changements

- **Souvent, dès le début**

- Réfléchissez comme un jardinier pas comme un maçon...

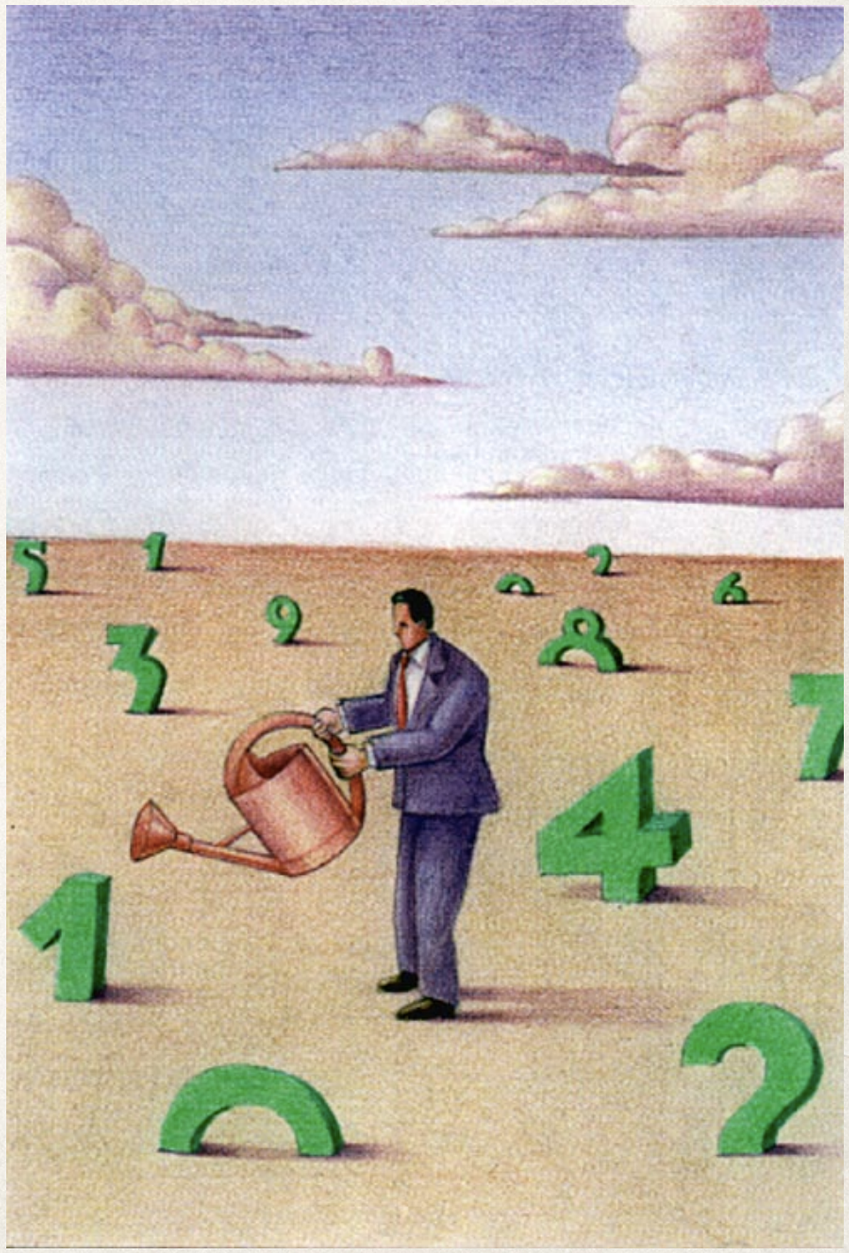


http://www.jhackens.com/2006/10/jardinier/

# «Refactoring» : comment ?

- Utilisez des outils pour identifier les changements (cf. cours Métriques)
- Utilisez des outils pour factoriser (Par exemple, Eclipse et les outils d'extractions de méthodes, ...)

- **Organiser le refactoring**
  - Planifiez, Mettez des priorités, Mémorisez les changements à faire
  - Soyez sûr de vos tests avant de refactoriser
  - Progressez pas à pas



http://www.jhackens.com/2006/10/jardinier/

# «Refactoring» : exemple

This Java code is part of a framework that will be used throughout your project. Refactor it to be more general and easier to extend in the future.

```
public class Window {
  public Window(int width, int height) { ... }
  public void setSize(int width, int height) { ... }
  public boolean overlaps(Window w) { ... }
  public int getArea() { ... }
}
```

# «Refactoring» : exemple

This case is interesting. At first sight, it seems reasonable that a window should have a width and a height. However, consider the future. Let's imagine that we want to support arbitrarily shaped windows (which will be difficult if the Window class knows all about rectangles and their properties). We'd suggest abstracting the shape of the window out of the Window class itself.

# «Refactoring» : exemple (2)

```
public abstract class Shape {
// ...
public abstract boolean
                overlaps(Shape s);
public abstract int getArea();
}
```

```
public class Window {
  private Shape shape;
  public Window(Shape shape) {
      this.shape = shape;
      ... }
  public void setShape(Shape shape) {
       this.shape = shape;
      .. }
public boolean overlaps(Window w) {
     return shape.overlaps(w.shape);
     }
public int getArea() {
      return shape.getArea();
     }
}
```

# «Refactoring» : exemple (2)

Note that in this approach we've used **delegation** rather than ***subclassing***: a **window is not a "kind-of" shape**—a window "has-a" shape. It uses a shape to do its job. You'll often find delegation useful when refactoring.
We could also have extended this example by introducing a Java interface that specified the methods a class must support to support the shape functions. This is a good idea. It means that when you extend the concept of a shape, the compiler will warn you about classes that you have affected. We recommend using interfaces this way when you delegate all the functions of some other class.


Sometimes my code
is like this.....
Don't know, what it does.
But i am scared to delete.

# Conclusion

- En route vers des designs patterns issus du GOF et quelques patterns d'architecture, mais les principes présentés restent toujours vrais et doivent diriger vos choix de mise en oeuvre.

# 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?