

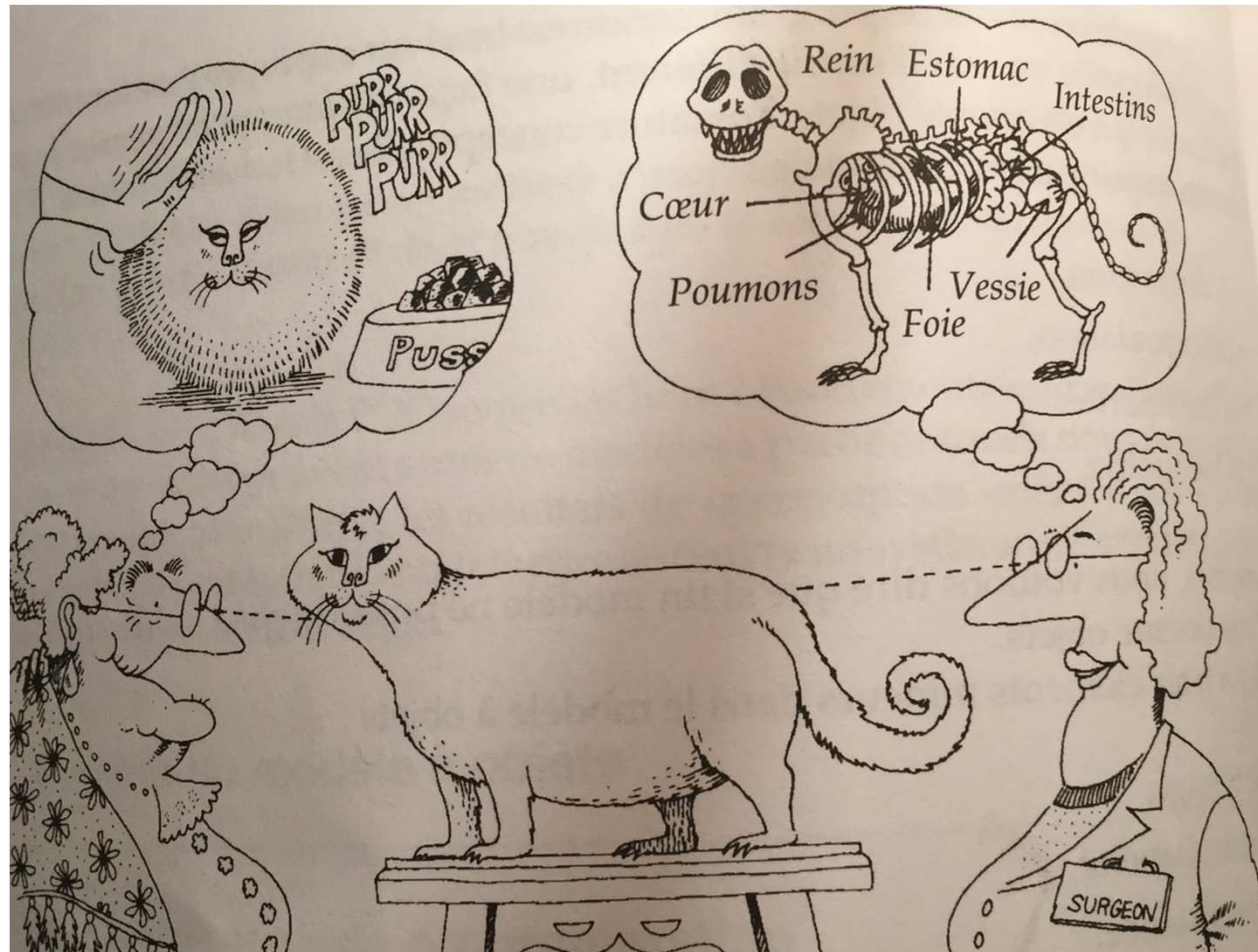
## Chapitre 2 : Abstraction, encapsulation et modularité

- 3 parties :
  - ➊ Abstraction
  - ➋ Encapsulation et masquage des données
    - Principe
    - Mise en œuvre
  - ➌ Modularité

# 1<sup>ère</sup> notion : l'abstraction

- Une abstraction fait ressortir les caractéristiques essentielles d'un type d'objet.
- Ces caractéristiques le distinguent de tous les autres types d'objets.
- Grâce à l'abstraction, le concepteur modélise les objets qui lui paraissent pertinents pour son problème (de son point de vue).
- Exemple : objet chat ?

# Extraits de Booch G., Analyse et Conception orientées objet, Addison Wesley, 1994.



Sources des images : Analyse&conception orientées objets - Grady Booch – Addison-Wesley

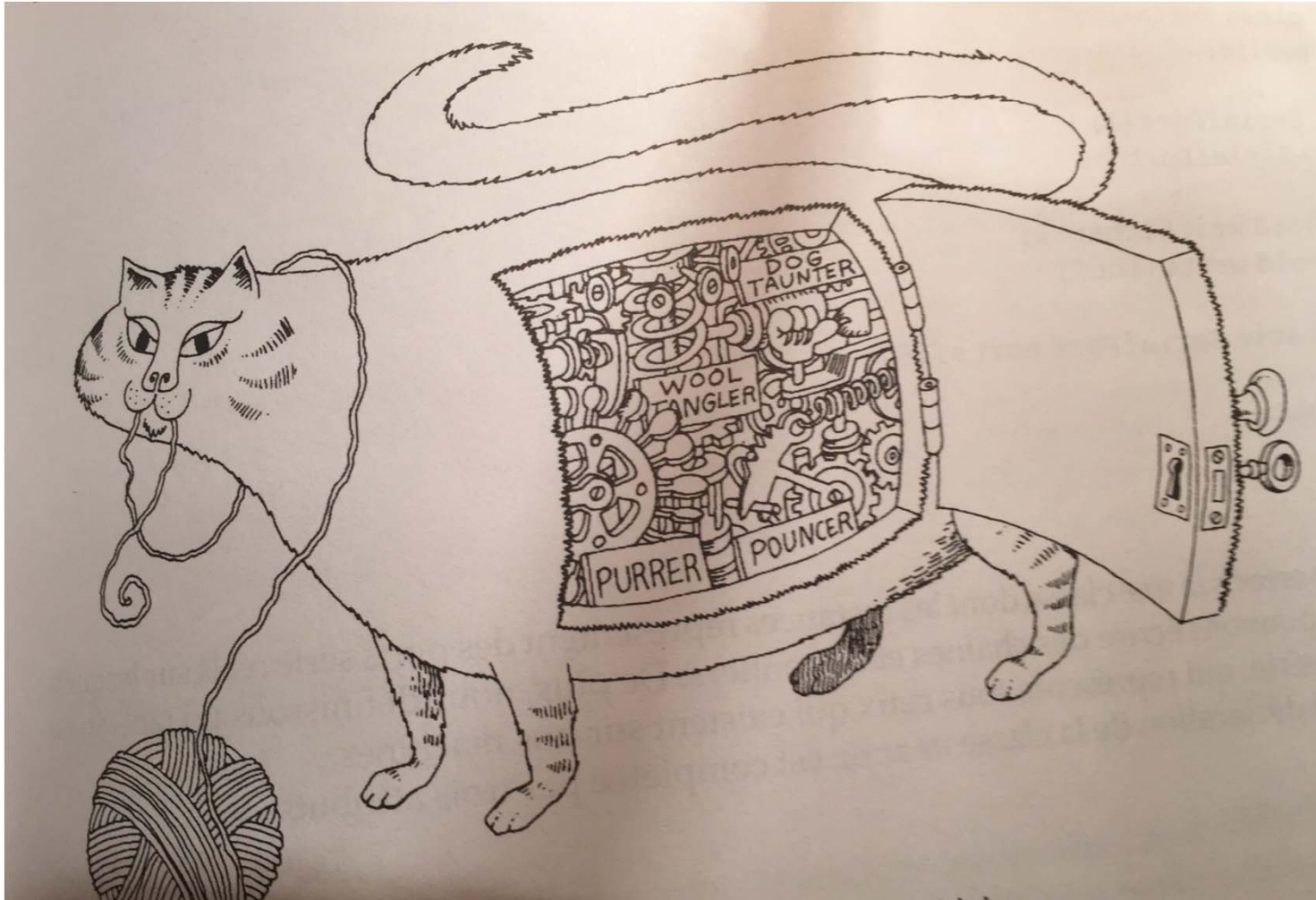
## 2ème notion : l'encapsulation

L'encapsulation est le procédé de séparation des éléments d'une abstraction (qui constituent sa structure et son comportement). Elle permet de dissocier l'interface contractuelle de la mise en œuvre d'un objet.

L'interface capture la vue externe de l'objet, qui renferme l'abstraction du comportement commun à tous les objets de même type.

La mise en œuvre renferme la représentation de l'abstraction ainsi que les mécanismes qui réalisent le comportement désiré.

# Interface versus implémentation



## 2ème notion : l'encapsulation

Le code et les données ne sont pas séparés mais regroupés dans un même module : la classe.

**TOUT** ce qui caractérise une famille d'objets est regroupé dans le modèle (classe ou prototype)

# En javascript (langage à prototype)

- Un objet a un attribut « prototype » qui donne le modèle de l'objet
- Pour une meilleure structuration du code, la définition du prototype est dans un fichier .js
- Ce fichier contient donc : le constructeur et les autres méthodes du prototype

## Exemple : classe Livre

```
// définition de Livre  
function Livre(nom, titre, nbPages)  
{this.nom=nom;  
  this.titre=titre;  
  this.nbPages=nbPages;  
  this.page=0; }  
  
Livre.prototype.lire= function()  
{ this.page=this.page+1; };
```



# Masquage des données

- L'encapsulation est habituellement obtenue grâce au **masquage des informations**.
  - La structure des objets (attributs) ainsi que le codage des méthodes sont cachés (non accessibles en dehors de la classe)
- L'accès aux attributs de l'objet ne peut être fait qu'au travers de méthodes.
- Les méthodes **publiques** définissent l'**interface** de l'objet

# En PHP (langage à classe)

```
class Mot {  
    // declaration des attributs  
    private $valeur;  
    private $longueur;  
  
    // declaration des methodes  
    function __construct($valeur)  
    {  
        $this->valeur = $valeur;  
        $this->longueur=0;  
    }  
  
} // fin de la classe Mot
```

public / private = modificateur d'accès



## private / public

- un élément déclaré **private** n'est accessible qu'à l'intérieur de la classe dans laquelle il est déclaré
- un élément déclaré **public** est accessible à l'intérieur et à l'extérieur de la classe

# En javascript, attributs et méthodes sont publics

On peut simuler le masquage des données avec les opérateur get et set :

```
function Chat(nom, poids) {  
    this._nom=nom;  
    this._poids=poids;  
    this._cri="miaouu";  
}
```

```
Object.defineProperty(Chat.prototype, 'nom', {  
  get: function() { return this._nom; } });
```

# Javascript : sucre syntaxique

- Javascript ES6 : ajout du mot-clé « class »
- Syntaxe basée sur les langages à classe
- Ne change RIEN à l'exécution : l'interpréteur produit du code basé sur le prototype

# Écriture avec le mot clé class

```
class Chat {  
  
    constructor(nom, poids) {  
        this.nom=nom;  
        this.poids=poids;  
        this.cri="miaouu";  
    }  
  
    crier() {  
        return this.cri;  
    }  
}
```

```
const chat1 = new Chat("Gros Minet", 3);  
console.log(chat1.crier());
```



## Les attributs sont publics

- Premier test :

```
const chat1 = new Chat("Gros Minet", 3);  
console.log(chat1.crier());
```

- Second test :

```
const chat1 = new Chat("Gros Minet", 3);  
chat1.cri="ouarf";  
console.log(chat1.crier());
```

# Opérateurs set et get

- Il est possible de simuler un attribut privé avec get et set :

```
class Chat {  
    constructor(nom, poids) {  
        ...  
        this._cri="miaou";  
    }  
  
    get cri() {  
        return this._cri;  
    }  
  
    set cri(value) {  
        this._cri = value;  
    }  
}
```



- Refaire le test :

```
const chat1 = new Chat("Gros Minet", 3);  
chat1.cri="ouarf";  
console.log(chat1.crier());
```

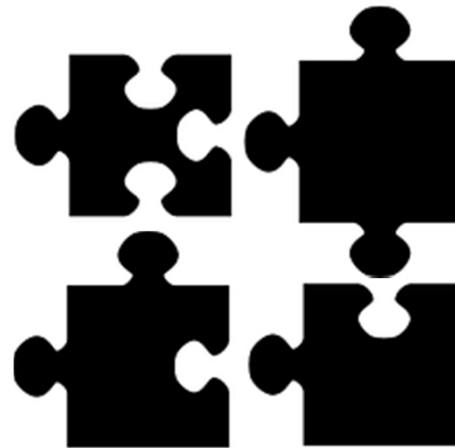
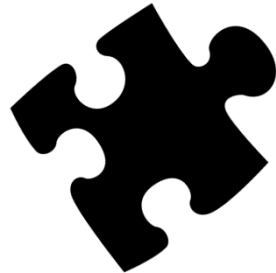
- Modifier le setter pour que le cri ne puisse pas être modifié

```
set cri(value) {  
    if (value.toUpperCase() == "MIAOU"  
|| value.toUpperCase() == "MIAOUU" )  
        this._cri = value;  
}
```

- Modifier la classe Chat pour :
  - qu'on ne puisse pas changer le nom du chat
  - qu'on ne puisse pas modifier le poids de plus ou de moins de 10%
- Ajouter la méthode manger(). Quand un chat mange, il grossit
- Ajouter la méthode jouer(). Quand un chat joue, il maigrit.

# Programmation modulaire

- Le code est découpé en **modules** = brique logicielle
- Les briques logicielles sont assemblées pour faire une application



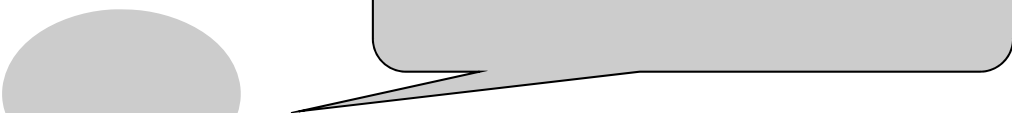
# Pourquoi la modularité

- développement en //
- débogage plus rapide
- code **réutilisable**
- code **extensible**



## Exemple : Mot / Page / Livre...

- une page :
  - a un numéro
  - est composée de mots
- on peut :
  - accéder à un mot de la page



déjà modélisé avec la  
classe Mot

→ des objets composés d'autres objets

# une application = des classes en relation

