PPOOGL

Modélisation objet UML

- Intro historico-sémantique
- Le vocabulaire de l'orienté-objet
- Quelques exemples de modélisation ratées et réussies
- Conclusion

Introduction

Où les étudiants apprennent avec consternation qu'ils vont devoir faire de la modélisation avant de se jeter sur leurs clavier.

Un peu d'histoire (1) : la préhistoire

Avant mon arrivée sur Terre, programmer c'était craignos :

- Fortran
- Basic
- plein d'assembleur
- et Lisp pour embêter les étudiants

Un peu d'histoire (2) : l'antiquité

Quand j'étais petit, la mode c'était Pascal :

- Programme = structures de données + algorithmes (Wirth)
- Typage fort et types structurés
- Code aussi structuré : *`Goto` considered harmful* (Dijkstra)
- Et donc sémantique compositionnelle

Et si on voulait des programmes efficaces malgré tous ces progrès, on avait C qui permettait de programmer proprement,
ou bien cochonnement. Selon le besoin.

C'est là aussi que les langages fonctionnels sont devenus civilisés, mais pas au point que les gens s'en servent, (à part pour programmer des éditeurs de texte à 10Mo).

Un peu d'histoire (3) : les temps modernes

Quand j'avais votre âge, tout-à-coup, la mode est passé aux langages orientés-objets :

- encapsulation : on ne voit d'un objet ce qu'on a besoin de voir
- héritage : permet l'abstraction et la réutilisation
- polymorphisme : permet malgré tout du code simple/lisible
- tout cela compatible avec les progrès précédents.

Pascal imposait la séparation données/algorithmes, les langages 00 imposent en plus la séparation fonctionnalité/implémentation.

Et si on veut des programmes efficaces malgré tous ces progrès, on a C++ qui permet de programmer proprement, ou bien cochonnement. Selon le besoin.

Même Caml est devenu orienté objet ! ? ! ? ? ! ! !

Fini les *langages*, bonjour les *méthodologies* de programmation

On dit souvent conception orientée objet au lieu de programmation :

- Depuis toujours, avant de se jeter sur leur clavier, les gens sérieux réfléchissent et font des petits dessins (modélisation)
  - Assembleur/Basic/Fortran : organigramme pour s'y retrouver dans les `goto`s.
  - Pascal : en plus, définition des structures de données, représentations graphiques des pointeurs...
  - Modèles de programmation exotiques : graphes de flots de données, automates, ...
- En principe, cette étape de modélisation est indépendante du langage.
- La suite de l'histoire fut de concrétiser ce "en principe".

Un peu de détente

Thus, an object-oriented analysis can be regarded as a form of syllogism moving from the Particular (classes) through the Individual (instances) to the Universal (control).

Ceci pour montrer que des bouquins entiers de philosophie ont été écrits sur la modélisation orientée objet, indépendamment de tout langage.

Un peu d'histoire (4) : les Balkans

Petite histoire de l'orienté objet :

- Dans les années 80, OO = UI (parfois AI)
- Fin des années 80 : Smalltalk (OO intégriste)
- Fin des années 80 : Booch écrit *Object Oriented Design* qui parle de ADA
- Explosion (effet de mode) au début des années 90. Langage à la mode : C++, dont la spécification est alors pleine de bugs.
- Plein d'outils de modélisation (OOSE, OMT, OBA, BON, MOSES, SOMA, ...) souvent accrochés à un langage.

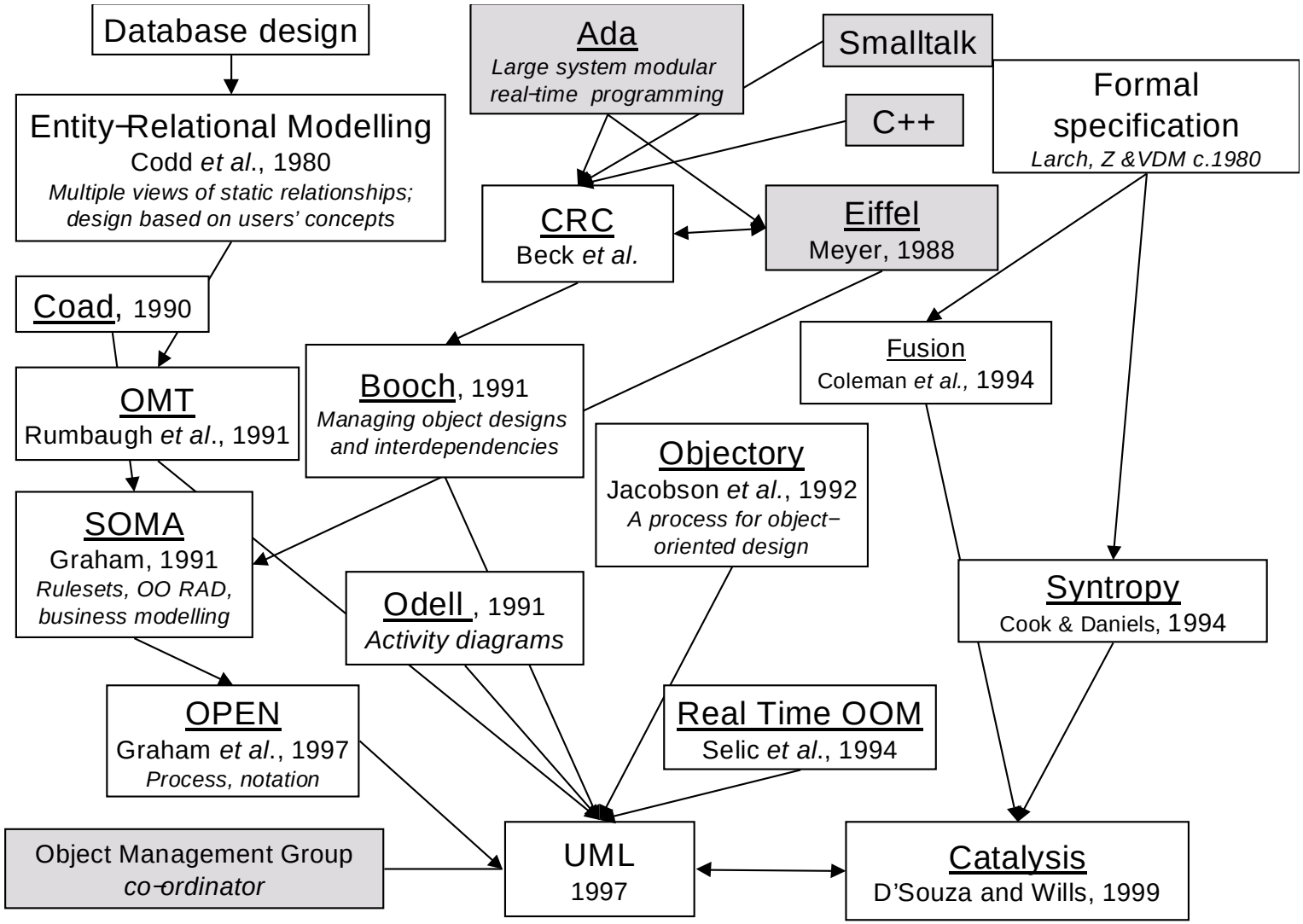
In 1994 there were over 72 methods or fragments of methods. The OO community soon realized that this situation was untenable if the technology was to be used commercially on any significant scale. They also realized that most of the methods overlapped considerably. Therefore, various initiatives were launched aimed at merging and standardizing methods.

Petit jeu : trouvez des significations crédibles de ces acronymes.

Un peu d'histoire (5) : le nouvel ordre mondial

Et c'est ainsi que naquit UML, l'*Unified Modelling Language*.

Les bonnes fées se penchent sur le berceau d'UML



Un peu d'histoire (5) : l'époque contemporaine

Après il y a eu d'autres langages objets, par exemple Java.

Mais ils n'inventent plus rien. Ils font juste tout un peu mieux...

Je ne vais donc pas vous apprendre Java (ni C++, ni SmallTalk, ni rien). Je vais donc vous expliquer les Grands Principes de l'Orienté Objet. Ensuite vous vous débrouillerez pour apprendre les langages que vous voudrez vous-mêmes, sur le tas.

Conclusion de l'intro : cékoi UML ?

- un prétexte facile pour rester dans les idées générales sans me plonger dans un langage particulier.
- Un langage graphique tout con pour représenter les notions OO
- Un standard pour communiquer des concepts 00 :
  - spécification (chef vers MIM)
  - travail collaboratif (MIM vers MIM)
  - documentation (MIM vers chef) qui n'y connait rien à Java++)
- Assez bien spécifié pour qu'on puisse en faire des outils
  - pour dessiner de l'UML (exemple sous Linux : `dia`)
  - ... qui exportent des squelettes de programmes (exemple : `dia2code`).
  - grâce à quoi vous économisez du boulot, car n'est-ce pas, vous devez toujours produire code + doc.

La vraie doc d'UML est (entre autres) sur `www.omg.org` (site du consortium *Object Management Group*) où vous constaterez que je vous ai bien simplifié les choses...

Le vocabulaire de l'orienté objet

- vocabulaire en français
- vocabulaire en anglais
- vocabulaire en UML

Objet

Un objet est composé de

- des attributs
  (parfois appelées *champs*, *variables*, *données membres*)
  $\approx$ les champs d'une structure de donnée des langages classiques
- et aussi des méthodes
  (parfois appelées *opérations*, *fonctions membres*)
  $\approx$ fonctions/procédures qui dépendent sémantiquement de l'objet.

Si vous n'avez pas compris le mot "sémantiquement",
la dernière phrase veut dire :
dont le bon sens dit qu'elles dépendent de l'objet.

Si vous n'avez pas de bon sens, je vous en donne dans la suite.

Encapsulation (1)

Pour chaque objet, il y a

- ceux qui le programment, et
- ceux qui l'utilisent (dans d'autres programmes).

Ce ne sont pas toujours les mêmes.

Encapsulation (2)

Pour chaque objet, il y a

- des attributs et des méthodes internes, utilisés pour le programmer
- des attributs et des méthodes externes, utilisés pour l'interfacer

Ce ne sont pas toujours les mêmes.

Encapsulation (3)

- Un objet a une interface par laquelle on le manipule, et que tout le monde connait. Cette interface fait partie de sa spécification. Elle est tout ce dont ont besoin les utilisateurs de l'objet.
- On a tout intérêt à cacher les sordides détails de l'implémentation d'un objet à ses utilisateurs :
  - ils en ont ainsi une vue plus simple, plus lisible ;
  - leur code ne risque pas d'interférer avec le code interne à l'objet ;
  - on peut changer l'implémentation d'un objet de manière indolore/transparente pour ses utilisateurs ;

Encapsulation (4)

Techniquement, l'encapsulation c'est très simple : un attribut ou une méthode sont

- soit publics, visibles par tout le monde, et moralement destinés à être accédés par tous les utilisateurs de l'objet ;
- soit privés, relatif au fonctionnement interne de l'objet, et cela ne regarde pas les utilisateurs.

Il y a en général au moins un niveau d'accès intermédiaire, appelé protégé. Il permet l'accès par du code "ami" : typiquement, les autres objets d'une même bibliothèque.
Remarque : On peut implementer cette philosophie dans n'importe quel langage, mais les langages OO apportent une garantie mécanique qu'elle est respectée. Enfin sauf Python.

Encapsulation (5)

Pour récapépeter, qui fait quoi :

- Le Chef donne la spécification de chaque objet, en décrivant
  - sa syntaxe publique (en UML par exemple)
  - sa sémantique (en langue naturelle jusqu'à nouvel ordre)
- Le programmeur responsable de l'objet bricole comme il veut ce qui est privé, tant qu'il respecte la spécification.
- L'utilisateur de l'objet (qui est aussi un programmeur) ne voit et n'a besoin que de ce qui est public. Il ne risque pas d'interférer avec le boulot du responsable de l'objet puisqu'il ne le voit pas.

Remarque : le Chef n'a pas besoin de savoir programmer.
Sinon il ne serait pas Chef.

En résumé

Les grandes questions qu'on se pose dans les trois paradigmes de programmation séquentiels :

- Imperatif : **Comment ?**
- Declaratif (fonctionnel ou logique) : **Quoi ?**
- Orienté Objet : **Qui ?**

Classe

Classe $\approx$ type d'objet.

Objet = instance d'une classe.
On parle des méthodes de l'objet.

En UML, les dessins sont presque les mêmes.

0 0 data-centrique ?

À une époque, une justification de l'OO était :
Les données sont ce qui est stable et important,
plus que les programmes.

Implantation dans SmallTalk, où toutes les fonctions/procédures sont des méthodes qui n'existent pas indépendamment d'un objet.

Une approche intégriste peut devenir contre-productive.
Par exemple, en SmallTalk, l'addition de deux nombres est une méthode d'un des nombres qui prend en paramètre l'autre.

À mon avis c'est idiot.

Approche par responsabilité

On se réveille, c'est le transparent important du cours

Au cours de la modélisation, on essaiera de respecter la philosophie suivante :

- Attribut d'un objet :
  ce qu'il est de sa responsabilité de savoir.
- Méthode d'un objet :
  ce qu'il est de sa responsabilité de faire.
- Si la responsabilité est celle de la classe, on dit que la méthode ou l'attribut est statique

Cela nous règle l'exemple de l'addition SmallTalk : l'addition est de la responsabilité de la classe des nombres, pas d'un objet de cette classe. C'est donc une méthode statique, qui a deux arguments.

Héritage (1)

Définition :

La classe `machin` hérite de la classe `truc` signifie tout simplement :
un `machin` est un `truc`.

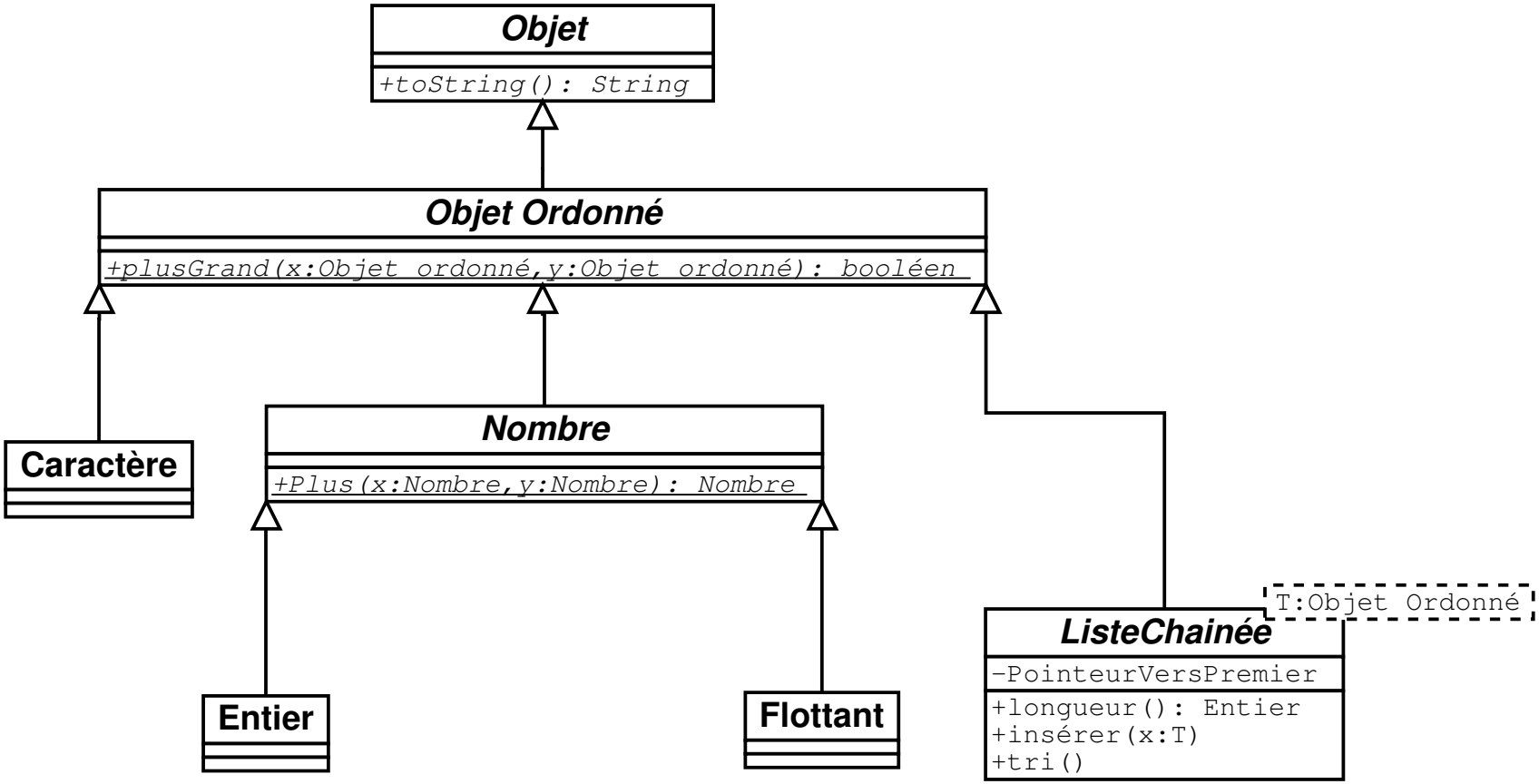
Une coccinelle est un coléoptère qui est un insecte qui est un animal qui est un être vivant, ce qui montre qu'on peut construire un graphe d'héritage, aussi appelé hiérarchie *is-a* ou hiérarchie de classes.

Héritage (2) : exemple

Comme tout le monde, je vais prendre un exemple qui va sur l'eau.

- Classe des objets
- Classe des objets munis d'un ordre total $<$
- Classe des caractères
- Classe des nombres informatiques
- Classe des entiers
- Classe des nombres en virgule flottante

Héritage(3) : en UML



Héritage (4) : pourquoi

L'intérêt c'est de définir les attributs et méthodes là où ils doivent l'être (approche par responsabilité)

- `toString()` est une méthode de tout objet, utilisée pour l'afficher sous forme textuelle. Par défaut elle affiche la valeur du pointeur vers l'objet.
- La plupart des classes redéfinissent (surchargent) `toString()` pour lui faire afficher quelquechose de plus informatif que la valeur du pointeur...
- ... mais un programme peut appeler `toto.toString()` quelque soit la classe de `toto`.
- Lorsqu'il y a plusieurs définitions de `toString` sur la branche de l'arbre d'héritage qui mène à la classe de `toto`, la définition du langage dit laquelle est appelée (typiquement la dernière), et comment appeler les autres si nécessaire.

C'est là qu'on remarque que finalement, utiliser `cout` en C++ c'est bien utiliser de l'orienté objet...

Héritage (5) : attributs et méthodes

Plus généralement, lorsque un `machin` est un `truc`,

- tous les attributs/méthodes de `truc` sont aussi des attributs/méthodes de `machin` : c'est pour cela qu'on dit que `machin` hérite tout de `truc`.
- `machin` peut aussi avoir des méthodes/attributs qui lui sont propres.
- `machin` peut redéfinir/surcharger/raffiner une méthode de `truc`, mais sans toucher à sa syntaxe externe (son prototype).

Finalement, dans un pointeur vers `truc` on pourra mettre un `machin` (puisque `machin` est un `truc`) et appeler pourtant toutes les méthodes d'un `truc`.

Héritage : subtilités

- notion de méthode abstraite : la surclasse déclare une méthode mais ne la définit pas. Elle oblige toutes ses sous-classes à la définir.
  Exemple : la méthode `cueillir()` de la classe `Fruits` : on cueille tous les fruits, mais pas de la même manière, et il n'y a pas de manière générique.
- notion de classe abstraite
- notion de polymorphisme, par exemple la méthode `toString()` de la classe des objets
- héritage multiple avec des conflits possibles : on verra plus tard.

(En UML on met en italique tout ce qui est abstrait.)

Type ou classe ?

Moralement c'est la même chose. Si vous voulez la petite nuance sémantique :

- Un type se met au singulier (une variable de type entier).
- Une classe se met au pluriel (la classe des nombres).

Informatiquement, Java a les deux : les types sont prédéfinis et fondamentaux (types numériques, char et booléen), et n'ont d'autre méthodes que celles fournies par le matériel. Du coup leur occupation mémoire est minimale. `String` est une classe. Il y a aussi une classe `Boolean` correspondant au type `boolean` mais on ne s'en sert jamais.

(cela se voit que je ne suis pas trop sûr de moi ?)

Objets de la “vraie vie” vs objets informatiques ?

(par “vraie vie” on entendra sans complexe un champ de bataille du futur avec des robots télépathes : c’est ce que le programme essaye de modéliser)

- La partie de la modélisation qui est facile c’est celle qui consiste à abstraire la “vraie vie”
- La partie qui est difficile c’est d’organiser vos objets informatiques (abstraire des objets abstraits...)
  (d’autant que c’est parfois arbitraire et vexatoire).
- Mais Caml vous a tellement habitué à l’arbitraire et au vexatoire que vous savez bien faire.

Le message, c’est qu’une bonne modélisation objet ce n’est pas (uniquement) d’utiliser une super classe `List` pleine d’héritage et de polymorphisme pour émuler le (non moins super) système de types de Caml. Je veux un bon modèle de la vraie vie.

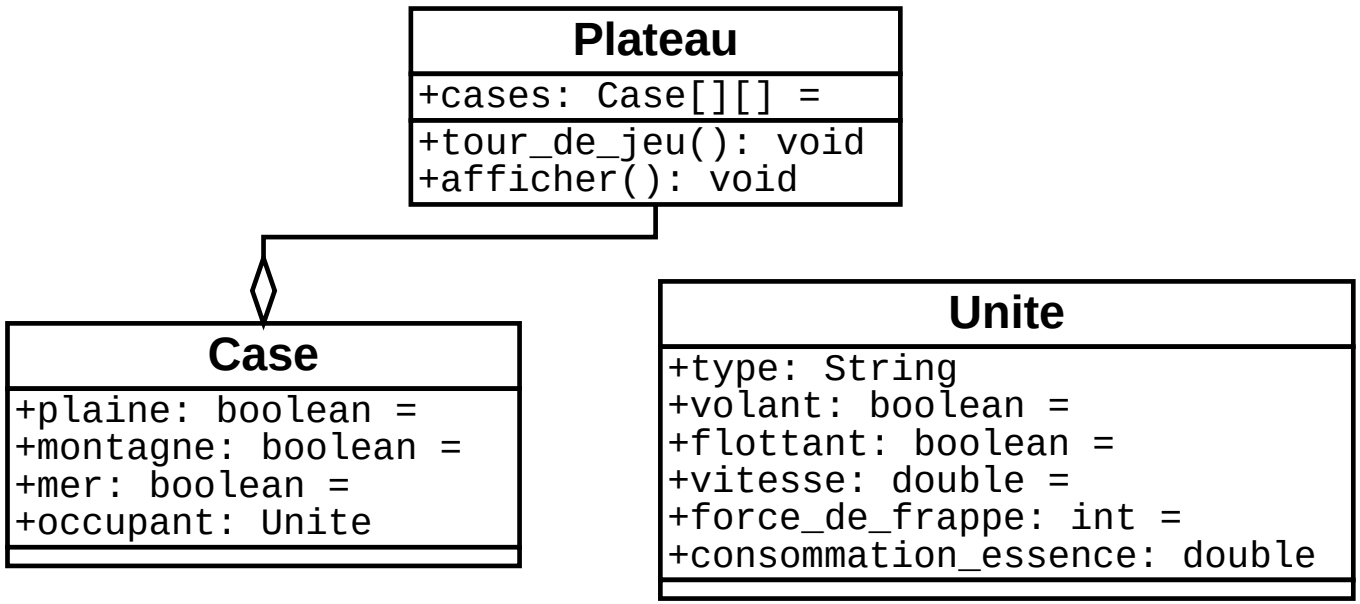
Exemples de modélisation OO

Où vos collègues de l'an dernier illustrent tout ce qu'il ne faut pas faire, en attendant que ce soit vous.

Un exemple facile

Un jeu de stratégie/plateau en réseau, dans lequel différents corps d'armée se bastonnent sur un terrain joliment quadrillé.

Et voila ce qu'ils m'ont fait



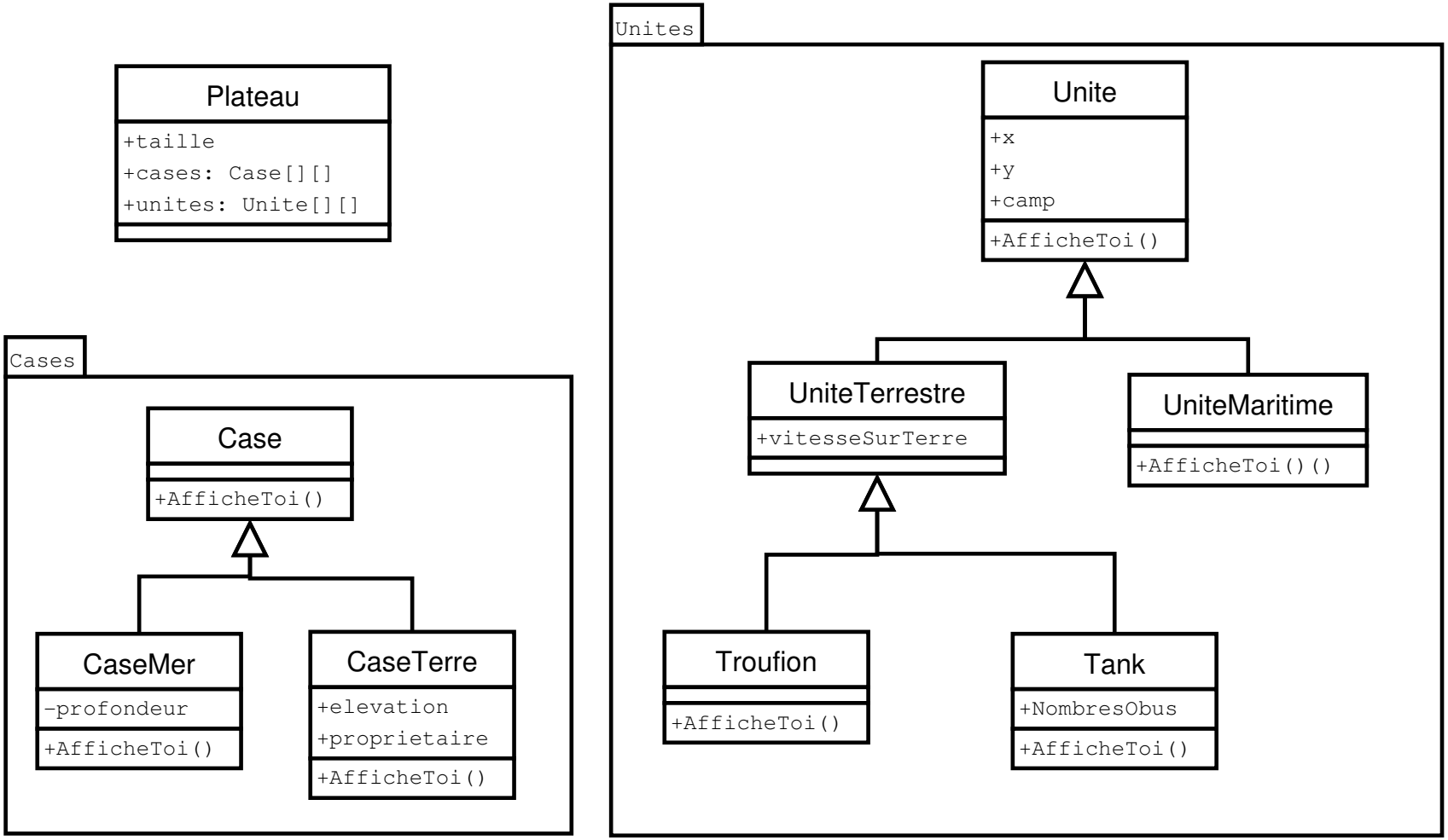
- écrire la procédure `afficher()`
- écrire la procédure `tour_de_jeu()`
- ajouter un véhicule amphibie qui a deux vitesses et deux consommations différentes suivant qu'il est sur terre ou sur mer.

Vous avez remarqué que `Case` et `Unité` n'ont pas de méthode ?

Une vraie modélisation objet

- Faire un graphe d'héritage pour les cases
- Faire un graphe d'héritage pour les unités
- Écrire la procédure d'affichage. L'affichage est de la responsabilité de qui ?
- Qui a la responsabilité de connaitre les différentes vitesses ?
- ajouter un véhicule amphibie qui a deux vitesses et deux consommations différentes suivant qu'il est sur terre et sur mer.

En UML, avec des paquetages



Les problèmes durs qui restent

- Deux unités se retrouvent sur la même case et se bastonnent. C'est de la responsabilité de qui ?
- Dans ce cas là, on n'y coupe pas à avoir plein de `if` dans le code.
- Toutefois on saura faire du code qui, en cas d'oubli d'un cas, a un comportement par défaut acceptable.
- Et ainsi le jour de la soutenance, F2D n'y verra que du feu.

Pour en finir avec l'UML

On a vu trois des boutons du mode UML de `dia`...
Il en reste juste 26 autres.
Ceux qui sont éventuellement utiles :

- Paquetages.
- Association, aggrégation et composition : variation sur *has a*.
- Tapez "UML" et l'un de ces trois mots dans Google et il vous en dira plus.
- Pas bien supportés par `dia2code` en tout cas.

Un exemple difficile

Un éditeur de partitions musicales proche de l'intuition des compositeurs.

- Qu'est-ce qu'un air ?
- Qu'est-ce qu'une mélodie ?
- Qu'est-ce qu'un accompagnement ?
- Qu'est-ce qu'un accord ?

Difficile car les objets sont tous abstraits, mais pas informatiques pour autant (pas des listes et des foncteurs).

Un exemple intéressant

Un moteur 3D animé par des modèles physiques

- Une partie à modéliser en copiant la vraie vie
- Une partie du modèle qui ne contient que des objets informatiques
- Partage des responsabilités pas toujours évident

Celui-là je veux bien que vous le refassiez cette année.

Conclusion

Les avantages de l'approche OO

- moins d'erreurs grâce à l'encapsulation
- moins de boulot grâce à la réutilisation

Pas convaincus, vous voulez toujours utiliser OCaml

- Vous allez commencer par venir pleurer “Si j’avais un vrai langage avec des fonctions d’ordre supérieur et des foncteurs, je pourrais modéliser tout vachement mieux qu’avec ces objets ripoux”. Ce ne sera pas vrai.
- Vous allez ensuite créer une super classe Liste, une super classe Fonction d’ordre supérieur, et une super classe Foncteur, et ensuite vous ferez du Caml en Java.
  Ce ne sera pas bien.
- Le but c’est qu’après ce cours, si vous avez un problème qui est naturellement OO, vous le voyiez et vous le traitiez en OO. Fût-ce en Ocaml.
- Tous les problèmes ne sont pas naturellement OO ! Mais je veillerai à ce que les vôtres le soient.

Mais laissons plutôt la parole à Xavier Leroy