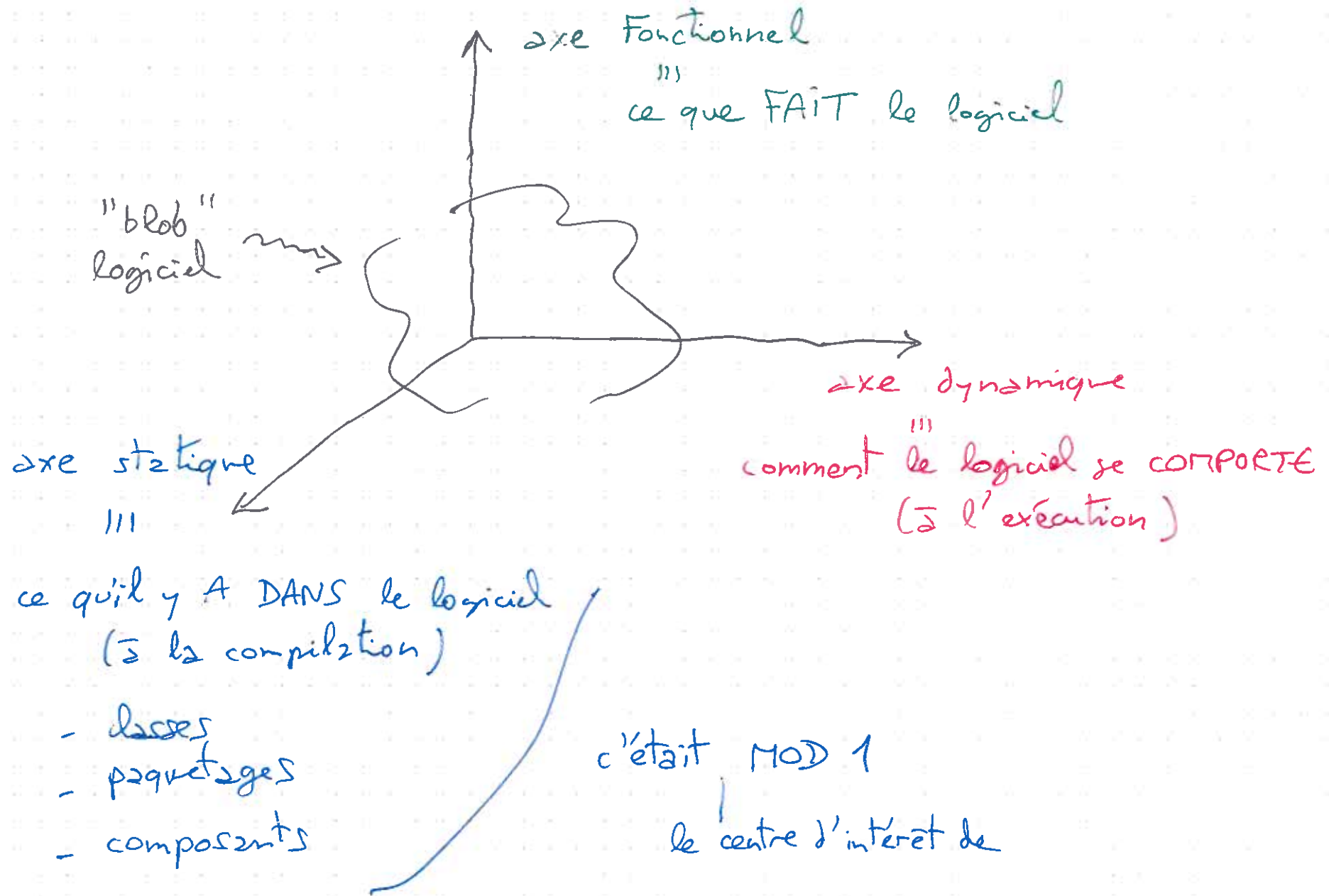


MOD2 = Cours modélisation avec des objets MOD version 2
2014
T. Gérard

c'est la suite de MOD1 !

→ planches 1/3

On peut décrire un logiciel suivant 3 axes :



Plan schématique du cours de MOD 2 :

- cours 1 : axe fonctionnel + un peu d'axe statique
- cours 2 : axe dynamique + un peu d'axe statique
- cours 3 : les 3 axes ensemble + un peu d'axe statique

La logique :

- le plus important = la description des fonctionnalités
- on en déduit la modélisation statique
- la modélisation statique doit pouvoir supporter
 - l'aspect dynamique du logiciel
 - les fonctionnalités attendues

on complète ce qui
a été vu en MOD 1

Rappels

- MODULE : - partie de logiciel cohérente
 - de taille raisonnable
 - bien découplée du reste du logiciel

→ un module fonctionnel est un ensemble de fonctionnalités répondant aux 3 items ci-dessus

les modules sont hiérarchiques :

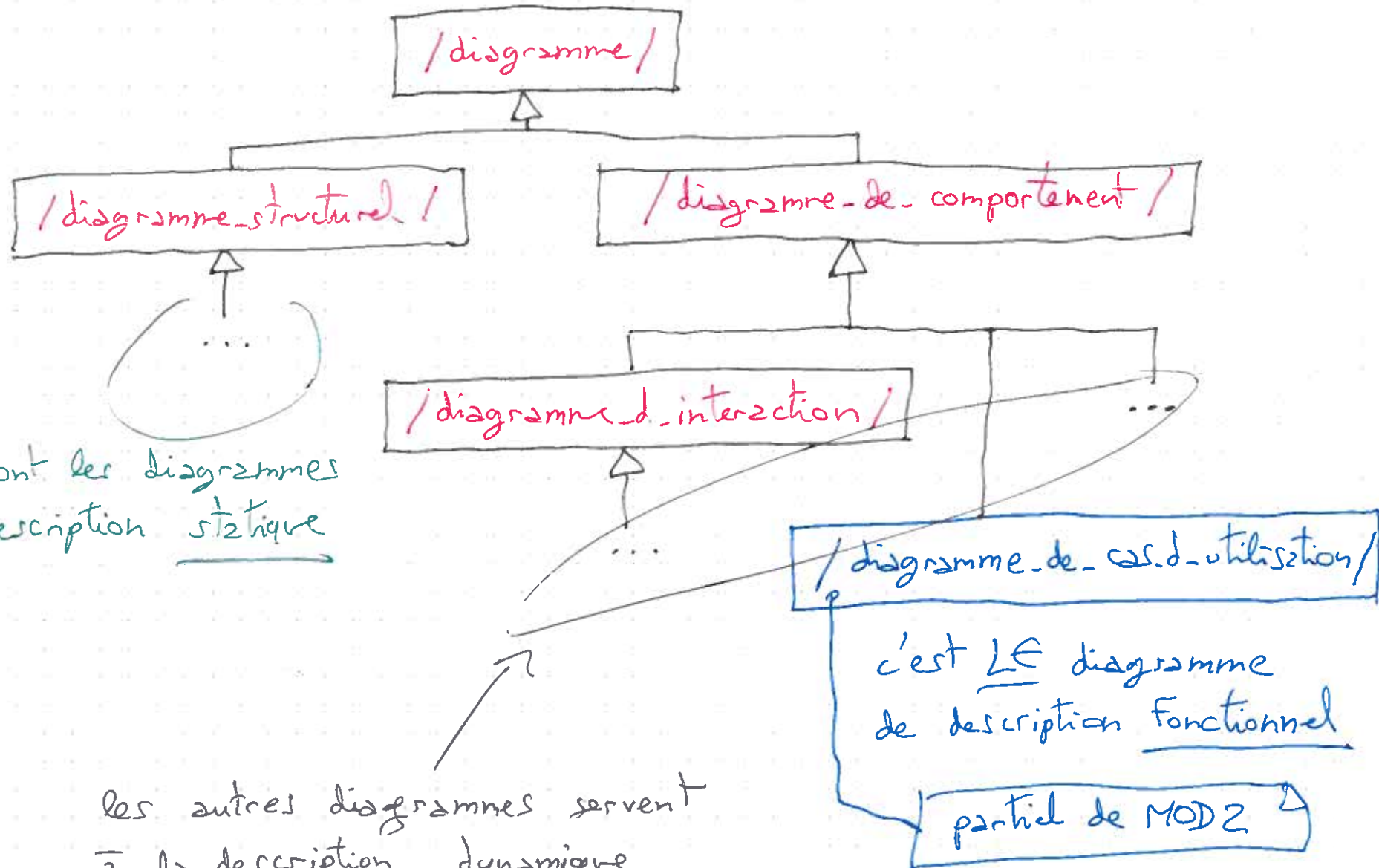
un module peut être décomposé en | sous-modules
| sous-parties

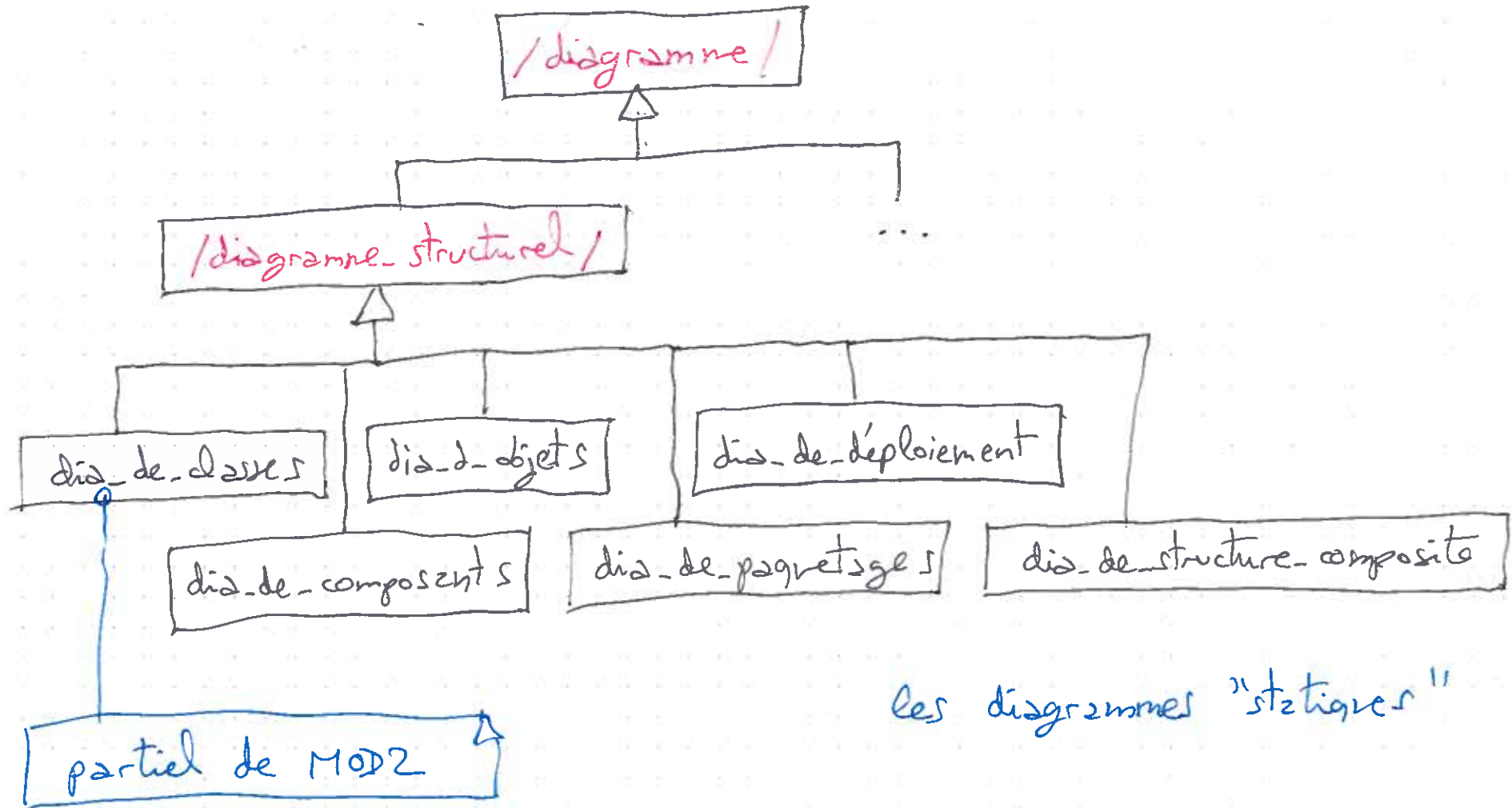
- MODÉLISATION : définition de la forme d'un logiciel
façonnage → description
suivant les 3 axes

- UML, pour Unified Modeling Language :

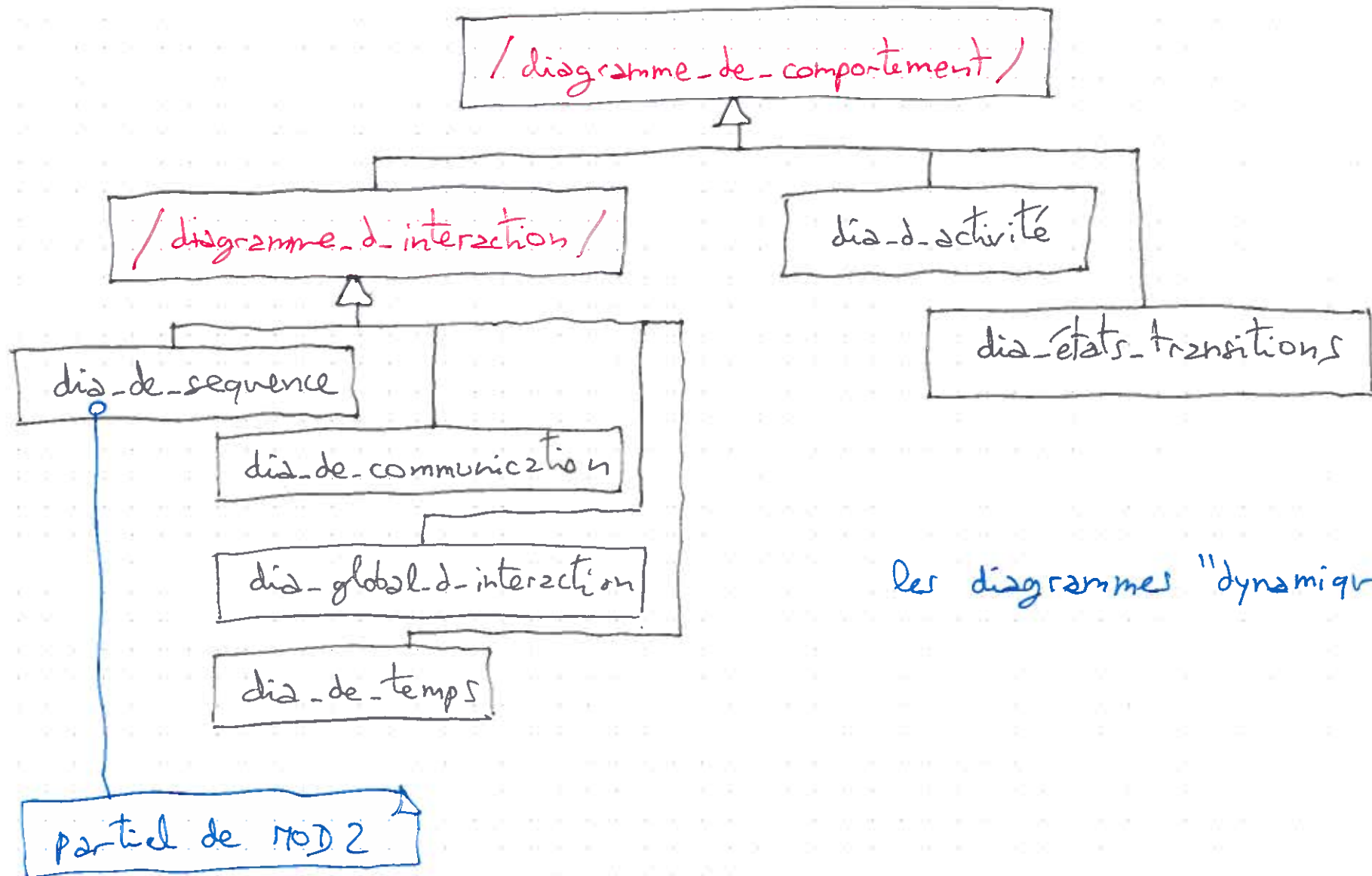
outil de description de logiciel, graphique et textuel

Il existe différents types de diagrammes UML





les diagrammes "statiques"



les diagrammes "dynamiques"

DESCRIPTION FONCTIONNELLE D'UN LOGICIEL

L'objectif: décrire les fonctionnalités d'un logiciel

⇒ chaque fonctionnalité

chaque module fonctionnel

les relations entre les fonctionnalités

et les modules fonctionnels

et... les acteurs

→ diagrammes de cas d'utilisation (CU)



Fonctionnalité dans son contexte

+ pour chaque CU, sa description en texte
et le scénario associé

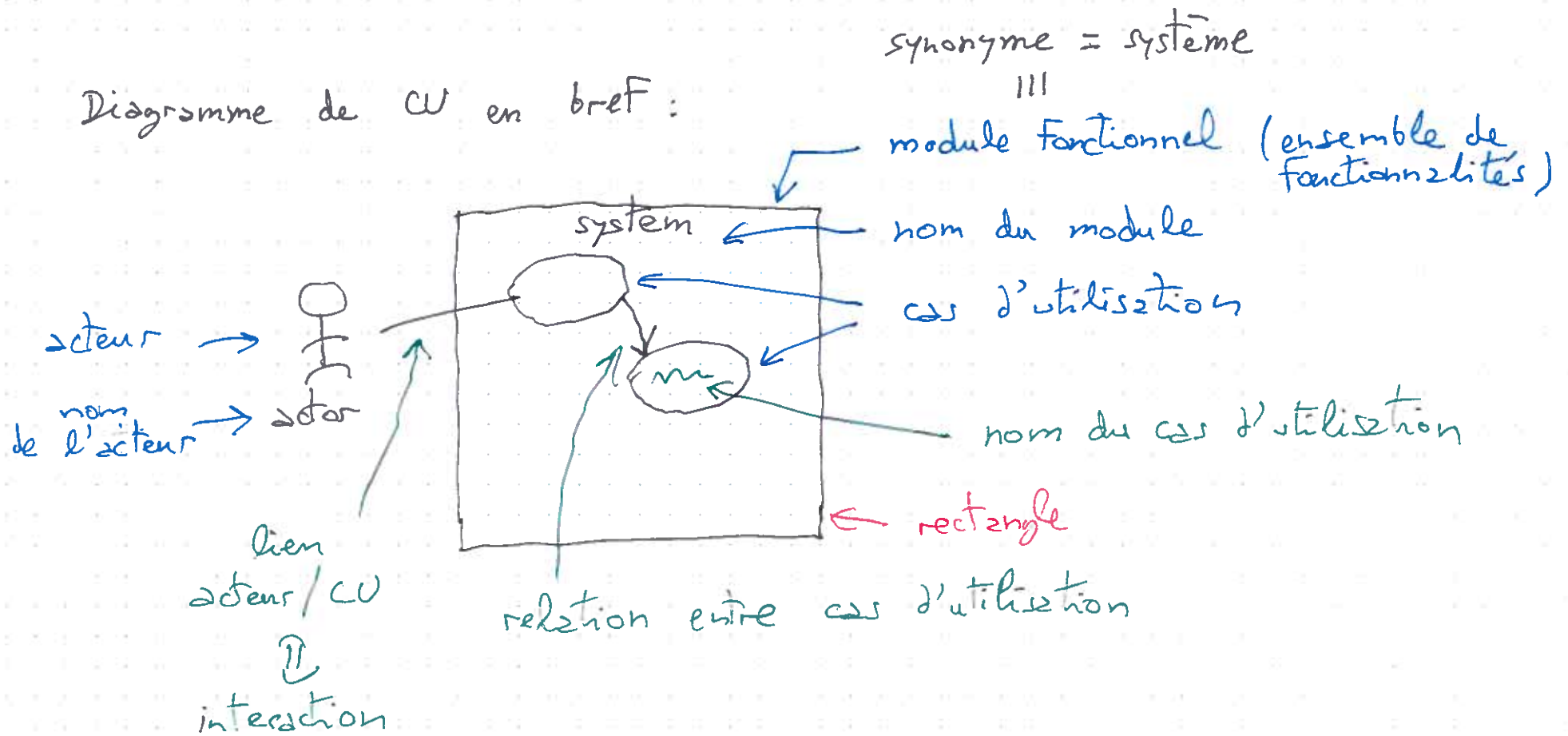


description dynamique

de la fonctionnalité →

(comment la fonctionnalité fonctionne
à l'exécution (ce qui se passe))

Diagramme de CU en bref :



- un acteur est :
- extérieur au système (un tiers)
 - en interaction avec au moins 1 CU du système
 - humain, logiciel, matériel ou autre

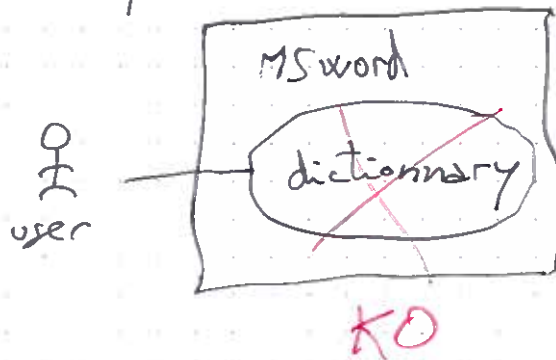
les systèmes sont hiérarchiques :

un système peut se décomposer en sous-systèmes.

Nommage des cas d'utilisation :

! doit contenir un verbe / désigne une action

exemple :



OK :

load a dictionary

OK :

spell check a word with a dictionary

OK :

add a new word in a dictionary

! un cas d'utilisation peut être purement "interne" au logiciel ou à un système \equiv non "visible" des autres systèmes et des acteurs .

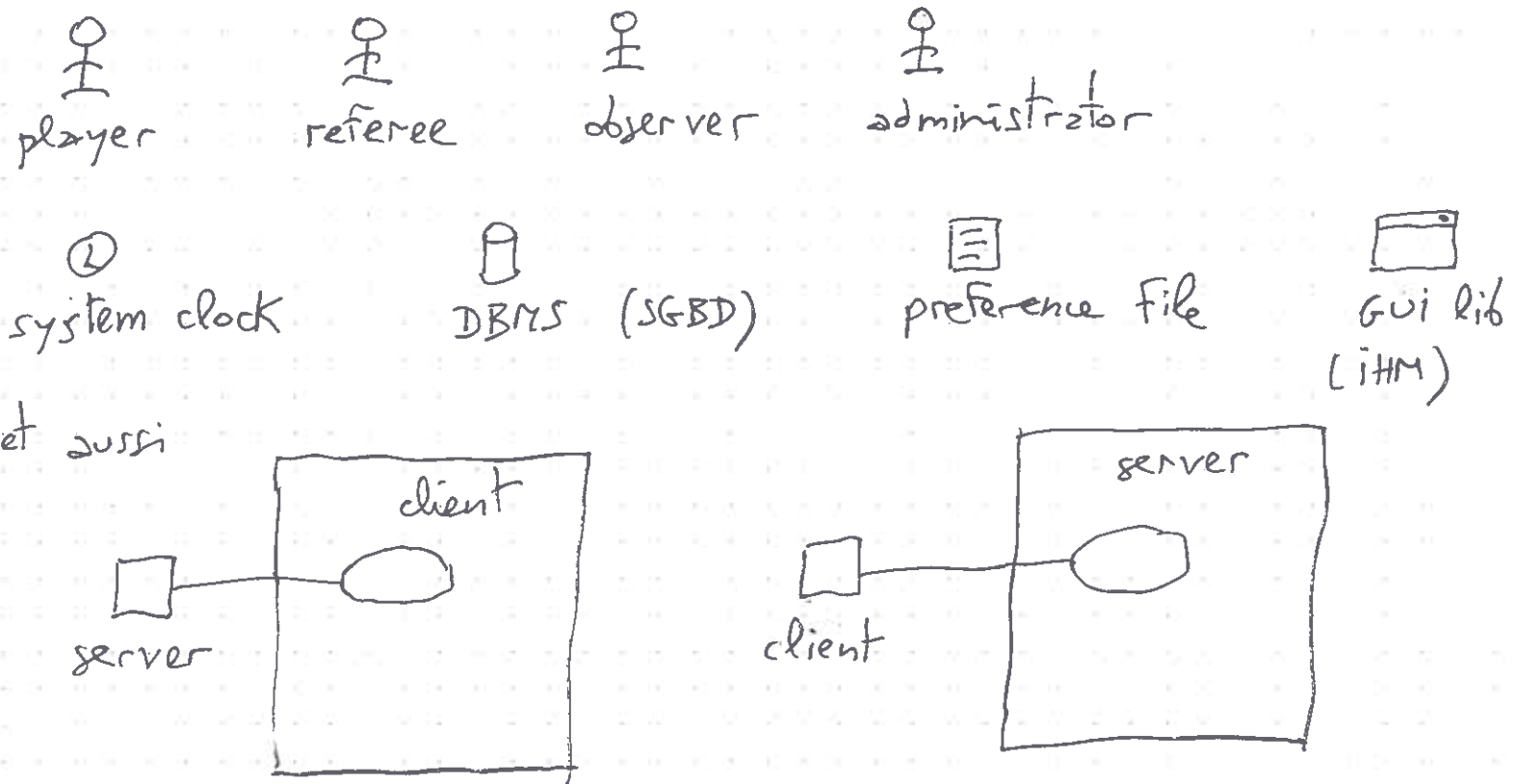
ex :


preload a sub-dictionary


(utile pour des raisons de performance)

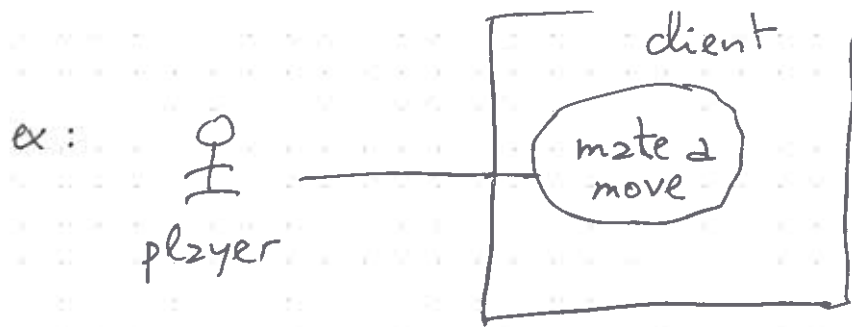
Exemples d'acteurs

Logiciel : jeu d'échecs en réseau

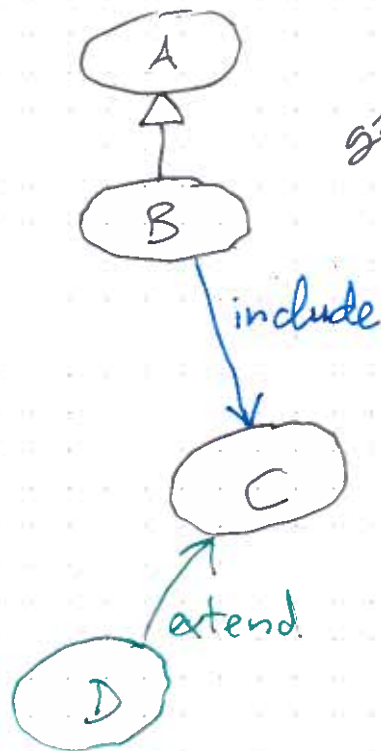


en revanche, ne pas considérer  **keyboard**
car interne à "GUI lib" et pas critique.

mg : pour certaines applis, l'  **arch** est critique...

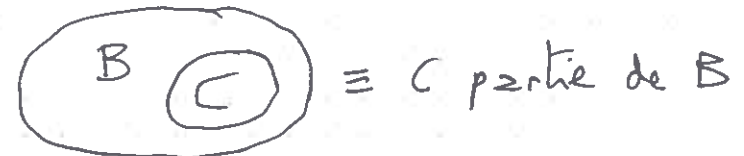


Relations entre cas d'utilisation



généralisation : B est une fonctionnalité A de nature particulière

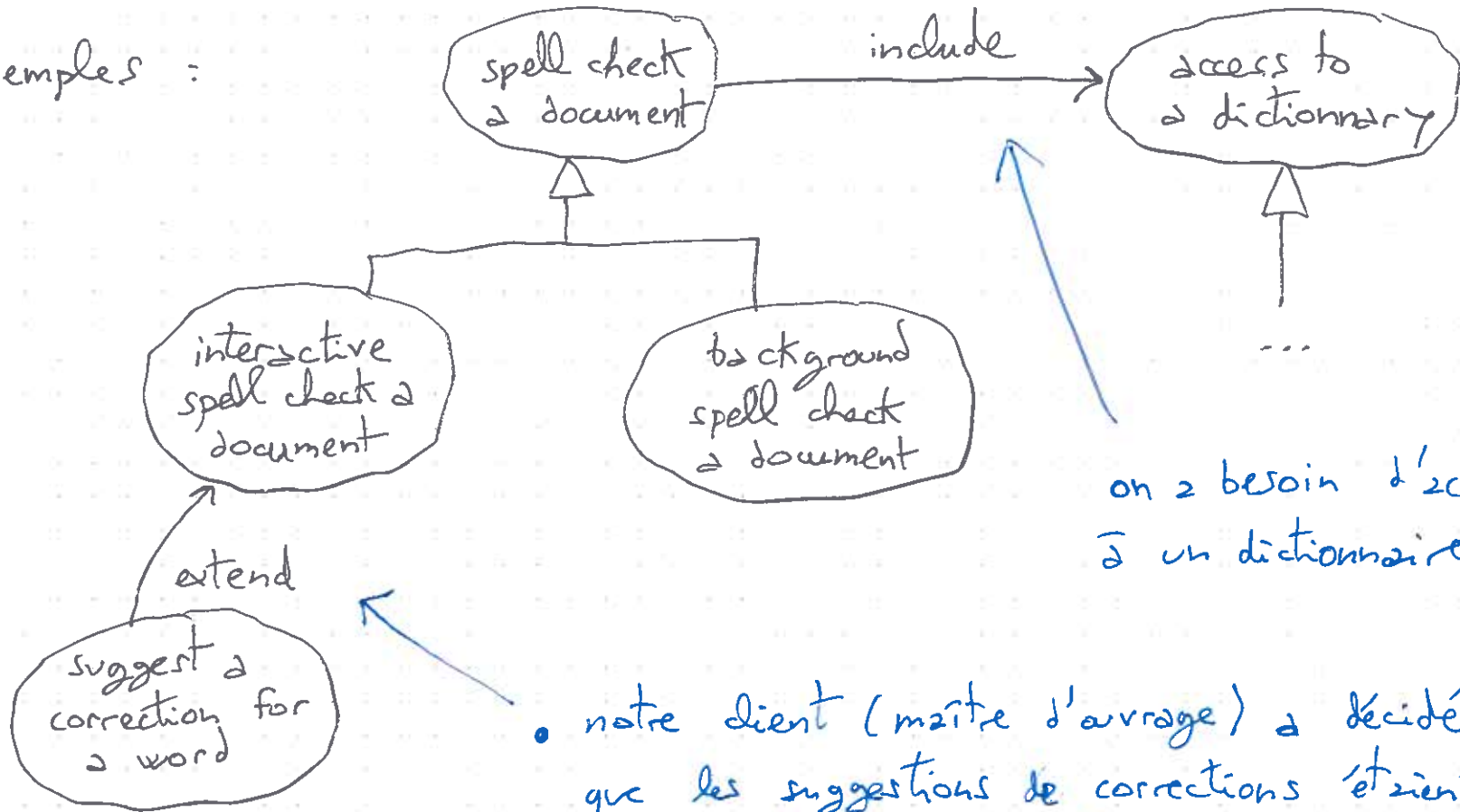
inclusion : B a besoin de C pour fonctionner
c'est comme si on avait



c'est une dépendance à-bas #include du langage C

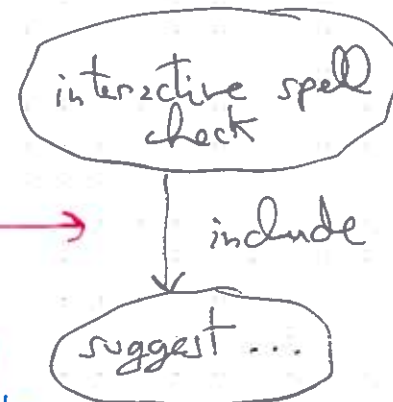
extension : D est une option "en plus" de C
(donc D a besoin de C pour fonctionner
et C peut fonctionner sans D)

examples :



on a besoin d'accéder à un dictionnaire

- notre client (maître d'ouvrage) a décidé que les suggestions de corrections étaient optionnelles \Rightarrow à cacher dans les préférences...
- il lui aurait pu décider l'inverse ; on aurait eu alors :



on en a donc bien besoin...

un piège ici, l'aspect dynamique n'intervient pas (exemple: le résultat d'un test (if) à l'exécution)

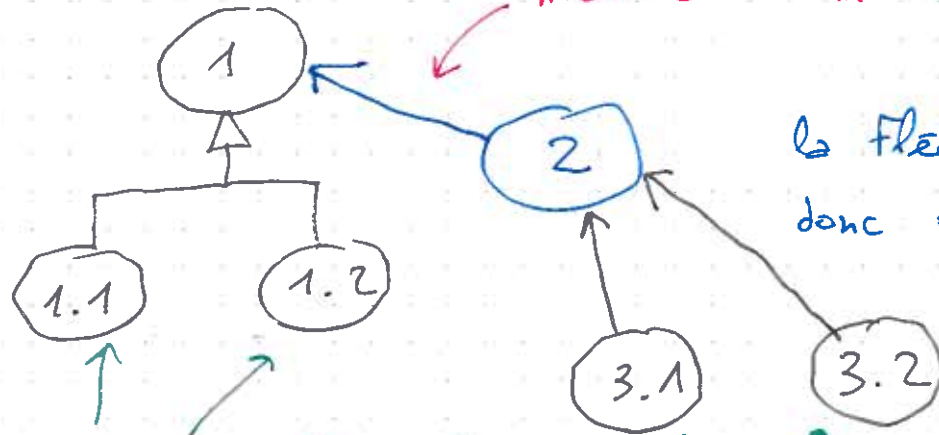
#include \Rightarrow "include" !

versus if(not isCorrect(word))

suggest_correction(word)

les diagrammes de cas d'utilisation permettent de planifier un projet

que ce soit
include ou extend



la flèche indique une dépendance
donc on planifie le CU1 avant le CU2

peuvent être planifiées
en parallèle

idem ici

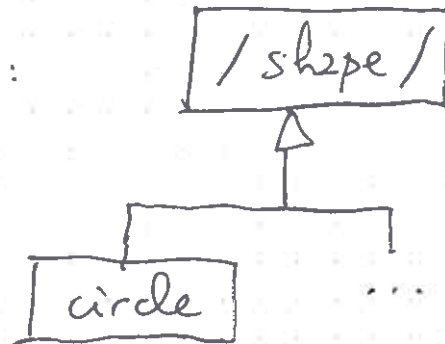
un BREAK statique

considérons un logiciel de dessin vectoriel

- on peut :
- dessiner des formes
 - effectuer des transformations géométriques
 - exporter son dessin
 - etc.

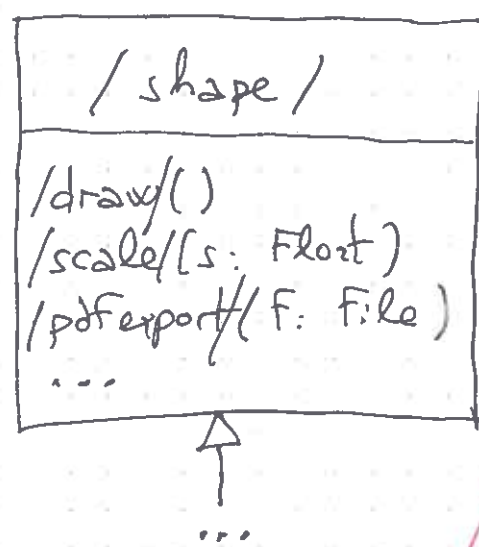
ce sont des
modules
fonctionnels
(différents)

on a déjà :



une hiérarchie de classes
est un module statique.

on pourrait être tenté par :



le module statique
mélange ici

plusieurs modules fonctionnels
différents !

(horrible ...)

rappel :

les modules DOIVENT ÊTRE
bien découplés

en totale opposition ici ...

cette très mauvaise modélisation est connue sous le nom
de "classe fourre-tout" (ici on a une hiérarchie Fourre-tout...)

ex: tout un programme java dans une classe Main !

on veut découpler :

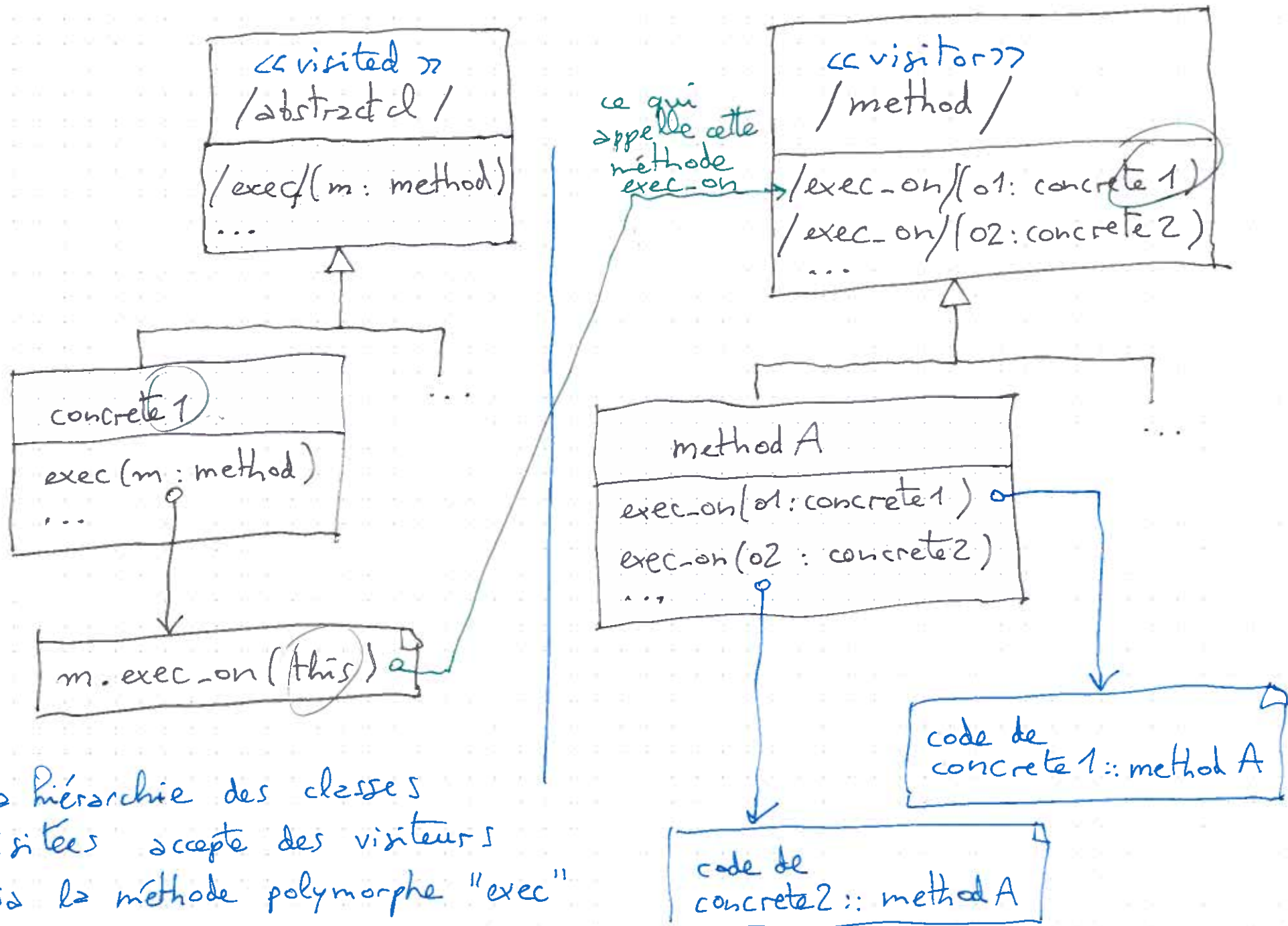
- une hiérarchie (module statique)
- des différentes fonctionnalités qui peuvent s'appliquer dessus (des modules fonctionnels)

pratiquement : circle :: scale, rectangle :: scale et consorts
doivent être hors de la hiérarchie de formes

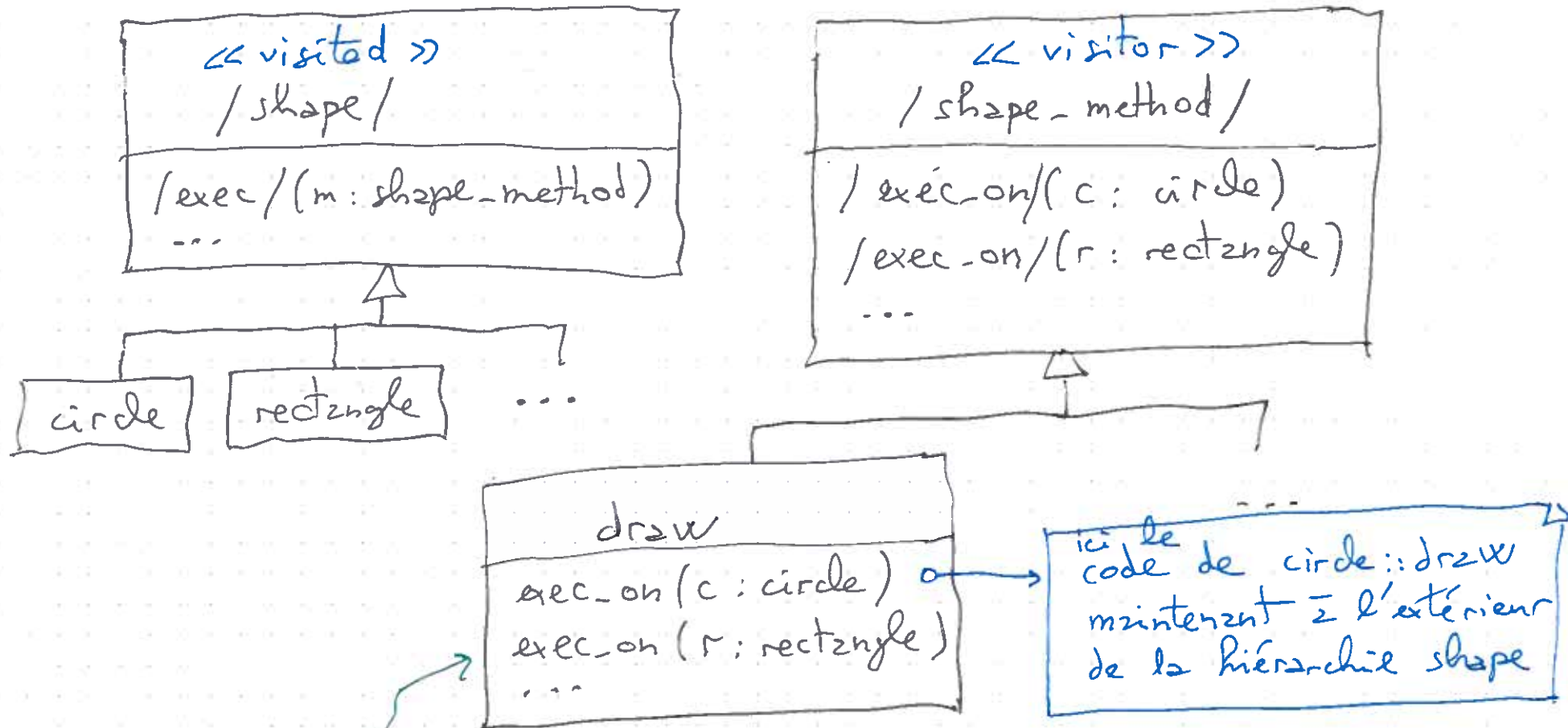
↙
Faisent partie du module
de transformation géométrique

= le module de
définition de formes

solution = le patron de conception "visiteur"
|
design pattern



Toutes les implémentations de methodA sont dans une classe.



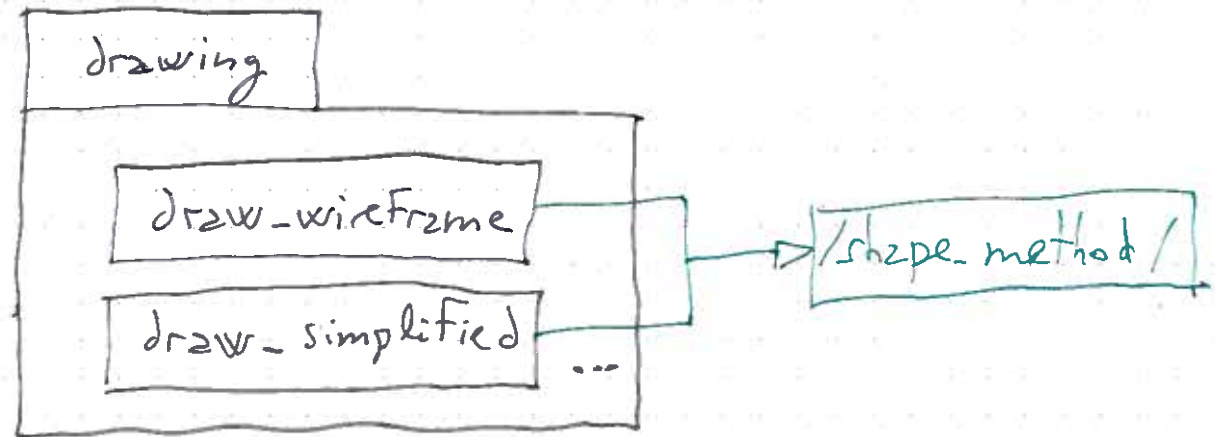
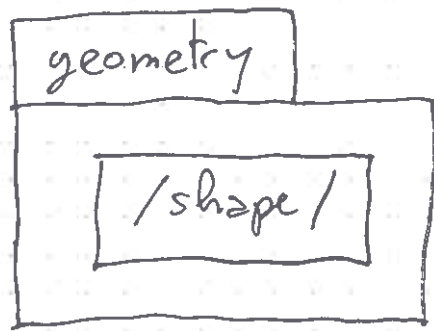
ici toutes les implémentations
de `shape::draw` sont
regroupées

on a donc "migré"

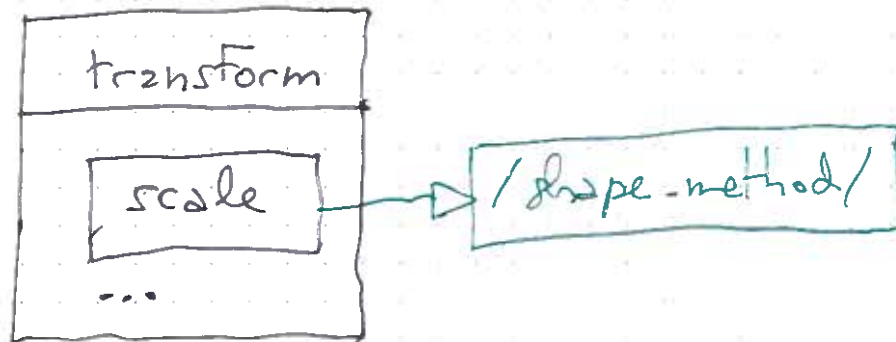
`circle::draw()` en `draw::exec-on(circle)`

en fait il existe plusieurs stratégies de dessin (fil de fer, etc.)
donc plusieurs (types de) méthodes `draw`...

Av final on a :



les modules statiques
sont maintenant
bien séparés



on ne fait plus
mais

```
s.drawWireframe()  
s.exec(new drawWireframe())
```

règle d'or :

les modules fonctionnels se traduisent
en modules statiques