Conduite de projet : La conception du logiciel

Mo Foughali
(IRIF, U-Paris) – foughali@irif.fr
(d'après transparents originaux de Yann Régis-Gianas et Aldric Degorre)

28 octobre 2021

“In the long history of humankind (and animal kind, too)
those who learned to collaborate and improvise
most effectively have prevailed”
- Charles Darwin

Séance de cours d'aujourd'hui

1. La conception d'un logiciel
2. La qualité d'un logiciel

La conception d'un logiciel

Le pourquoi, le quoi et le comment

Le pourquoi.

Toute construction humaine significativement complexe à besoin d'être pensée avant d'être réalisée.

Le pourquoi (suite).

En programmation, on parle de **spécification**.

En développement et ingénierie logicielle (dont la programmation est **une** composante, cf. CM1), on parle de conception.

La conception est donc un terme plus générique, englobant la spécification.

Les différents avantages de la conception ont été entamés en expliquant ceux de la spécification en CM1.

Voici les deux questions importantes de la phase de conception :

Quoi faire ? et Comment le faire ?

- L'**analyse fonctionnelle** répond à "Quoi faire".
- Les **conceptions architecturale et algorithmique** répondent à "Comment le faire".

Question : où se trouve la spécification (point de vue programmation) ci-dessus ?

La conception (le comment).

Comment concevoir un logiciel ?

En respectant le principe de **responsabilité unique** :
Je dois pouvoir décrire en une phrase la responsabilité de chaque **composant** et chaque **algorithme**. La contribution de chaque composant/algorithme dans la résolution du problème doit être claire.

Décomposition en problèmes élémentaires

L'élémentarité est un élément-clé de la maximisation des **propriétés de qualité** du logiciel :
la modularité, la maintenabilité, la simplicité, l'extensibilité, la réutilisabilité, la vérifiabilité, ...

Ce n'en est cependant pas le seul. Quoi d'autre ?

```
if (s.measurement.vel._present &&
    !std::isnan(s.measurement.vel._value.wx)) {
  (s.measurement.intrinsic ?
   (*context)->filter.measure.iw() :
   (*context)->filter.measure.w()) <<
    s.measurement.vel._value.wx,
    s.measurement.vel._value.wy,
    s.measurement.vel._value.wz;

  if (s.measurement.vel_cov._present) {
    double *c = s.measurement.vel_cov._value.cov;

    (s.measurement.intrinsic ?
     (*context)->filter.measure.iw_cov() :
     (*context)->filter.measure.w_cov()) <<
      c[9],  c[13], c[18],
      c[13], c[14], c[19],
      c[18], c[19], c[20];
  } else {
```

```
sds.p() = s.p() + ds.segment<3>(0);
sds.v() = s.v() + ds.segment<3>(6);
sds.w() = s.w() + ds.segment<3>(9);
sds.a() = s.a() + ds.segment<3>(12);

return sds;
```

La conception architecturale

Architecture logiciel

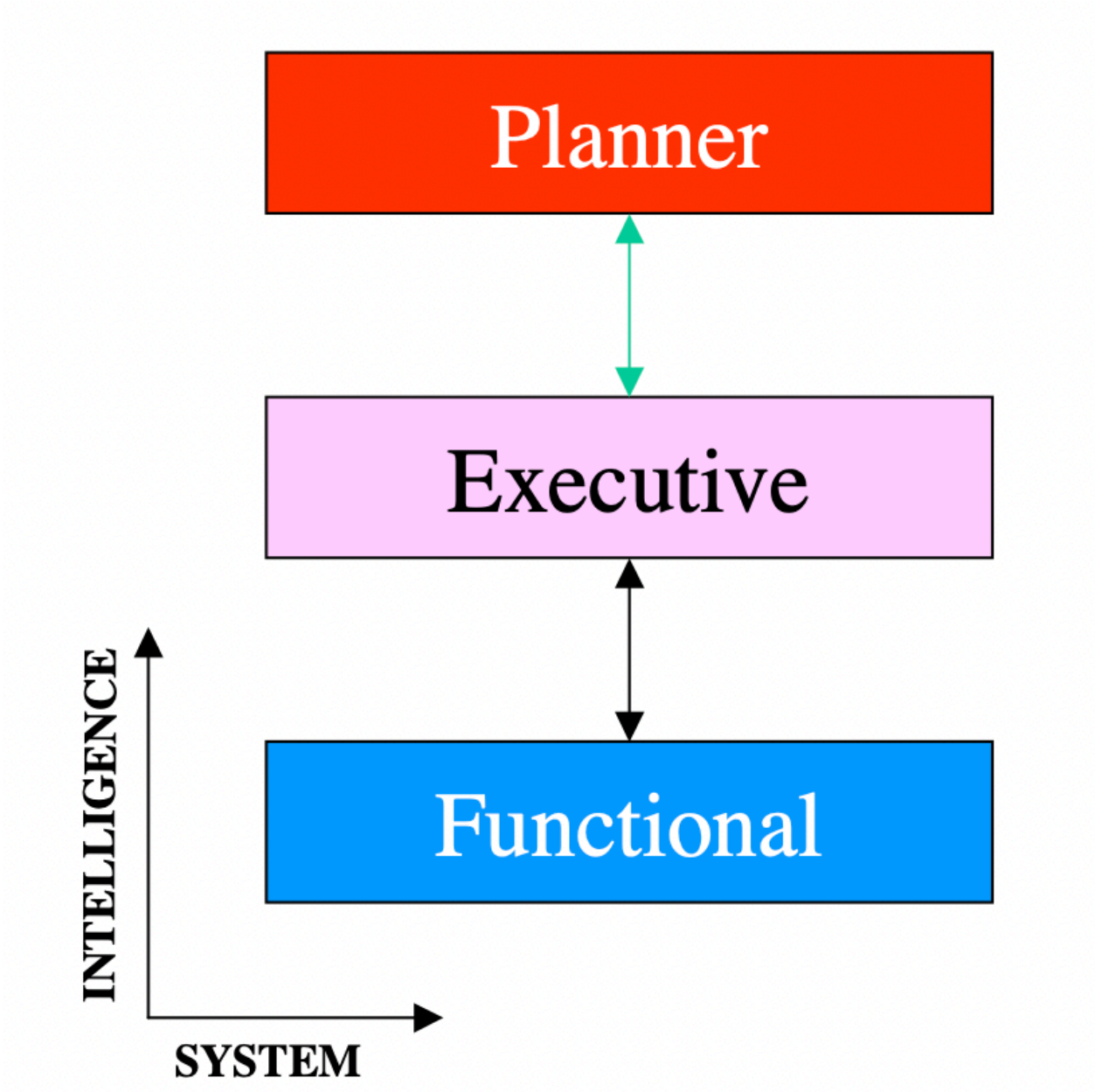
L'architecture d'un logiciel est l'ensemble des éléments qui le composent, leur responsabilité et leur interrelation.

C'est une sorte de "vue d'avion" du logiciel pour comprendre quels sont ses composants principaux et pourquoi ces composants sont effectivement une solution au problème posé par le logiciel.

Les grands styles architecturaux

- Architecture hiérarchique
- Architecture par couches
- Architecture centrée sur les données
- Architecture par flots de données
- Architecture par objets
- Architecture par services

Robots autonomes : l'architecture par couches CLARAty (NASA)



Robots autonomes : l'architecture par couches CLARAty (NASA) [2]

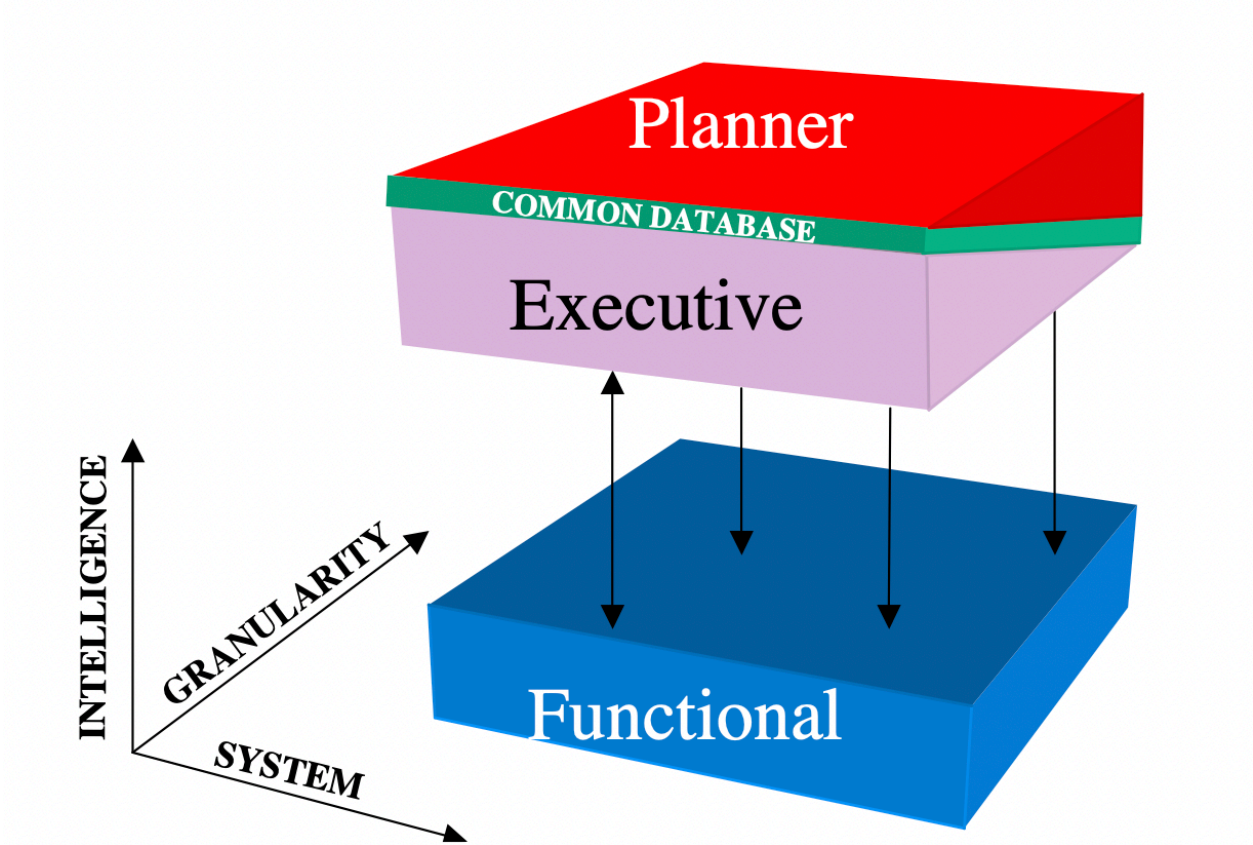


Figure 2. Proposed two-layer architecture.

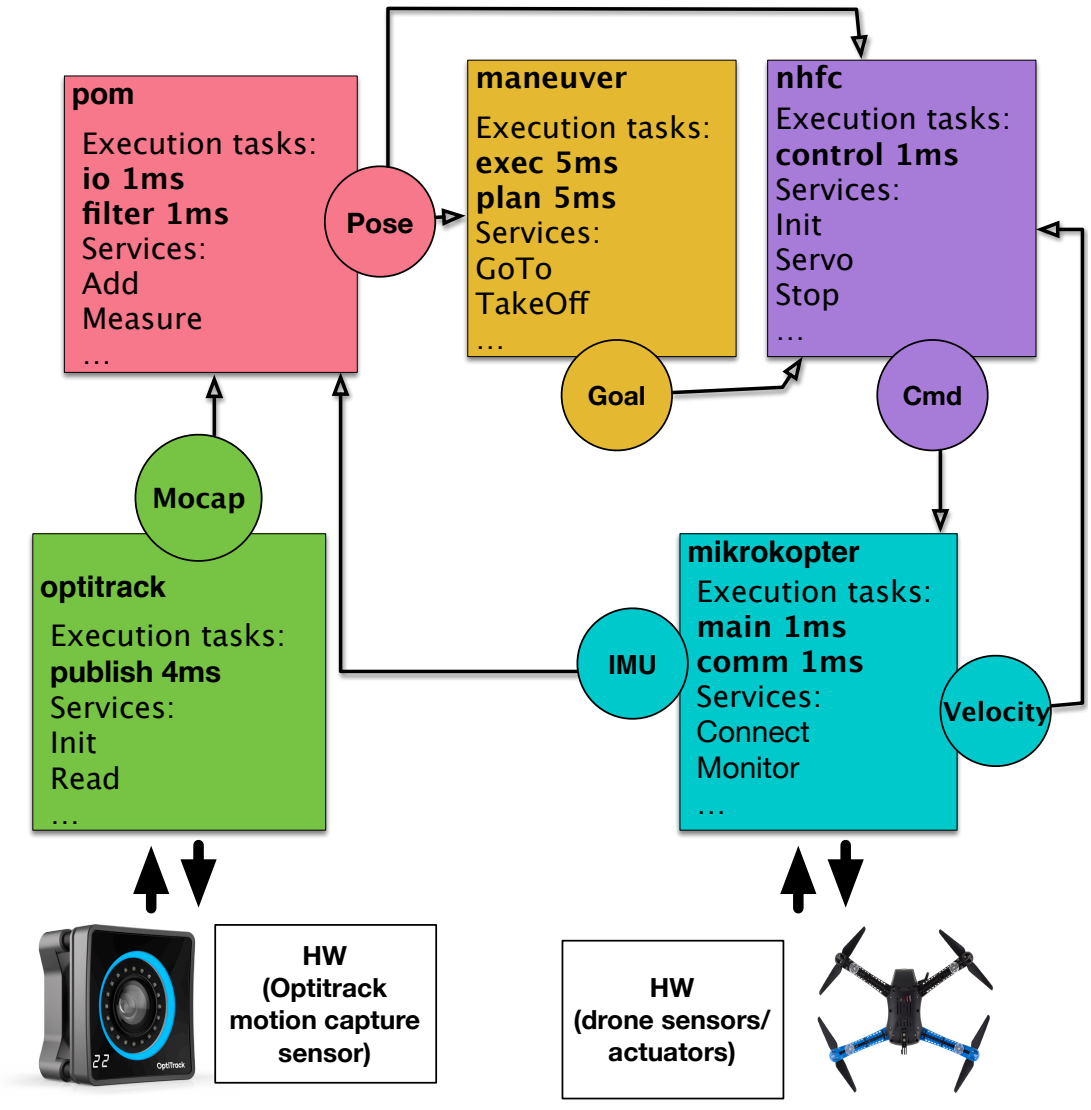
2. R. Volpe et al. The CLARAty architecture for robotic autonomy. IEEE Aerospace. 2001

Conception architecturale de `visulog`

Quel est le style architectural suivi par `visulog` ?

Conception architecturale vs. conception algorithmique

Qu'est-ce que la conception algorithmique ? Comment garantir le principe de responsabilité unique pour les deux types de conception ?



La qualité logicielle

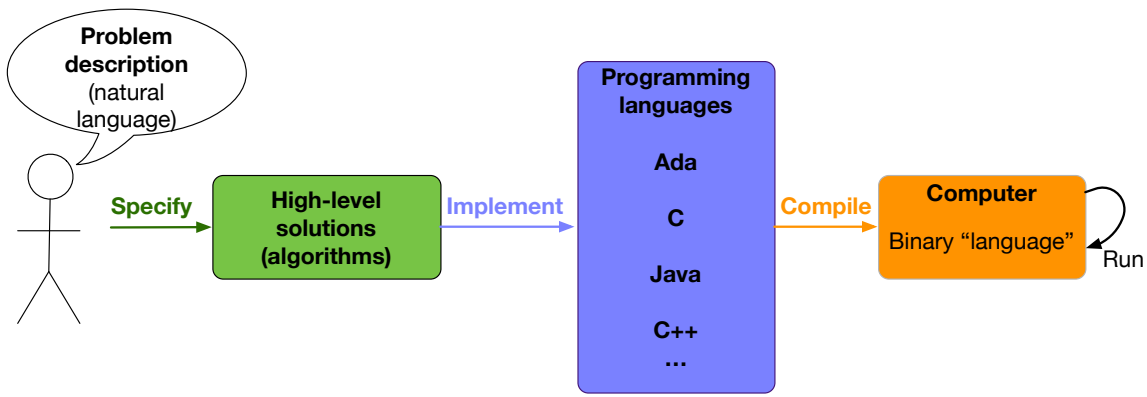
Selon vous, quels sont les critères qui font qu'un logiciel est de bonne qualité ?

ISO 9216

La norme ISO/CEI 9126 fixe les critères suivants :

- La capacité fonctionnelle :
  le logiciel répond aux exigences de fonctionnalités.
- La fiabilité :
  le logiciel fonctionne sous les conditions d'usage normales établies.
- L'utilisabilité :
  le logiciel est adapté à ses utilisateurs.
- L'efficacité :
  le logiciel utilise raisonnablement les ressources allouées à son fonctionnement.
- La maintenabilité :
  le logiciel peut évoluer.
- La portabilité :
  le logiciel peut être transféré d'un environnement à un autre.

Réutilisabilité vs. portabilité



Comment améliorer la qualité d'un logiciel ?

- La capacité fonctionnelle :
- La fiabilité :
- L'utilisabilité :
- L'efficacité :
- La maintenabilité :
- La portabilité :

Comment améliorer la qualité d'un logiciel ?

- La capacité fonctionnelle :
  $\Rightarrow$ Des tests, des preuves, de la documentation.
- La fiabilité :
- L'utilisabilité :
- L'efficacité :
- La maintenabilité :
- La portabilité :

Comment améliorer la qualité d'un logiciel ?

- La capacité fonctionnelle :
- La fiabilité :
  $\Rightarrow$ Idem, mais dans un contexte différent (c'est à dire ?)
- L'utilisabilité :
- L'efficacité :
- La maintenabilité :
- La portabilité :

Comment améliorer la qualité d'un logiciel ?

- La capacité fonctionnelle :
- La fiabilité :
- L'utilisabilité :
  $\Rightarrow$ Utiliser des béta-testeurs (voire des "lambda").
- L'efficacité :
- La maintenabilité :
- La portabilité :

Comment améliorer la qualité d'un logiciel ?

- La capacité fonctionnelle :
- La fiabilité :
- L'utilisabilité :
- L'efficacité :
  $\Rightarrow$ Des tests de charge, de montée de charge.
- La maintenabilité :
- La portabilité :

Comment améliorer la qualité d'un logiciel ?

- La capacité fonctionnelle :
- La fiabilité :
- L'utilisabilité :
- L'efficacité :
- La maintenabilité :
  $\Rightarrow$ Ecrire le code pour qu'il soit compréhensible. Réduire sa complexité. Utiliser des algorithmes et des composants élémentaires !
- La portabilité :

Comment améliorer la qualité d'un logiciel ?

- La capacité fonctionnelle :
- La fiabilité :
- L'utilisabilité :
- L'efficacité :
- La maintenabilité :
- La portabilité :
  $\Rightarrow$ Utiliser des technologies portables. Bien isoler les parties du système qui sont dépendantes de l'environnement.

Le développement dirigé par les tests

Qu'est-ce qu'un test ?

Un **test** est un programme qui vérifie une propriété d'un autre programme sur une entrée fixée.

- Un test n'est donc pas une preuve (mais les preuves coûtent cher).
- On appelle **batterie de tests** l'ensemble des tests d'un logiciel.
- Un test permet d'améliorer la qualité d'un logiciel.
- Une batterie de tests permet la **non-regression** de la qualité du logiciel.
- Un test peut aussi servir à **reproduire un bug**.

Qu'est-ce qu'un test ?

Un **test** est un programme qui vérifie une propriété d'un autre programme sur une entrée fixée.

- Un test n'est donc pas une preuve (mais les preuves coûtent cher).
  $\Rightarrow$ Ce n'est pas une excuse pour ne pas faire de preuves (on verra cela en M1).
- On appelle **batterie de tests** l'ensemble des tests d'un logiciel.
- Un test permet d'améliorer la qualité d'un logiciel.
- Une batterie de tests permet la **non-regression** de la qualité du logiciel.
- Un test peut aussi servir à **reproduire un bug**.

Qu'est-ce qu'un test ?

Un **test** est un programme qui vérifie une propriété d'un autre programme sur une entrée fixée.

- Un test n'est donc pas une preuve (mais les preuves coûtent cher).
- On appelle **batterie de tests** l'ensemble des tests d'un logiciel.
  $\Rightarrow$ On parle aussi souvent d'une **campagne de tests (testing campaign)**. Qu'est-ce que c'est ?
- Un test permet d'améliorer la qualité d'un logiciel.
- Une batterie de tests permet la **non-regression** de la qualité du logiciel.
- Un test peut aussi servir à **reproduire un bug**.

Pour résumer, un test est une **connaissance exécutable sur le logiciel**.

Classification des tests

- **test unitaire** : vise à exercer le fonctionnement d'une unité logique du logiciel. Dans un langage orienté objet, on écrit généralement un test par objet.
- **test d'intégration** : vise à exercer la bonne interaction entre les composants du logiciel.
- **test système** : vise à exercer le fonctionnement du logiciel dans son ensemble. On peut vérifier beaucoup de choses : fonctionnalités, performances, gestion des exceptions, montée en charge, utilisabilité ...
- **test d'acceptation** (ou recette) : vise à assurer que le déploiement chez le client se passera bien. On distingue les tests d'acception "en usine" et ceux fait chez l'utilisateur.

Distinction “boîte noire” et “boîte blanche”

- Un test **boîte noire** ne suppose rien sur le fonctionnement interne du composant logiciel testé : il n’utilise que son interface.
- Un test **boîte blanche** s’autorise à s’appuyer sur le fonctionnement interne du composant.
- Boîte noire : robuste mais moins précis que boîte blanche.
- Boîte blanche : plus difficile à maintenir mais généralement nécessaire.

Propriétés

- Une **propriété** décrit les “traits” du logiciel.
- On distingue **propriétés fonctionnelles** et **propriétés non fonctionnelles**.
- Propriété fonctionnelle : liée à “ce que fait le logiciel/programme”.
- Propriété non fonctionnelle : liée à “comment le logiciel/programme le fait”.
- Parmi les manières dont les propriétés fonctionnelles peuvent être décrites, on trouve la spec préconditions-invariants-postcoditions :
  - les **préconditions** décrivent les hypothèses sur les entrées.
  - les **postconditions** décrivent les garanties sur les sorties.
  - les **invariants** décrivent des propriétés toujours vraies qui font que le système fonctionne bien.

Lien entre test et propriétés

- Un test (fonctionnel) peut servir à vérifier la bonne implémentation d'une conception vis-à-vis d'une ou plusieurs propriétés sur une entrée fixée.
- Le test doit fournir des entrées validant la précondition.
- Le test doit vérifier que la sortie valide la postcondition.
- Un test en boîte blanche peut aussi vérifier que les invariants internes du composant/algorithme reste valide entre chaque interaction avec ce dernier.

Comment écrire de bons tests ?

- Un test est indépendant des autres tests.
- Il faut traiter les cas standards mais surtout les cas limites.
- Il faut essayer d'écrire de petits tests.
- Les tests doivent être "simples", facile à comprendre.

Comment écrire de bons tests ?

Choix de tests par partitionnement

- Partitionner le domaine des entrées en sous-domaines.
- Choisir une entrée par sous-domaine.
- Tester sur chaque entrée choisie.

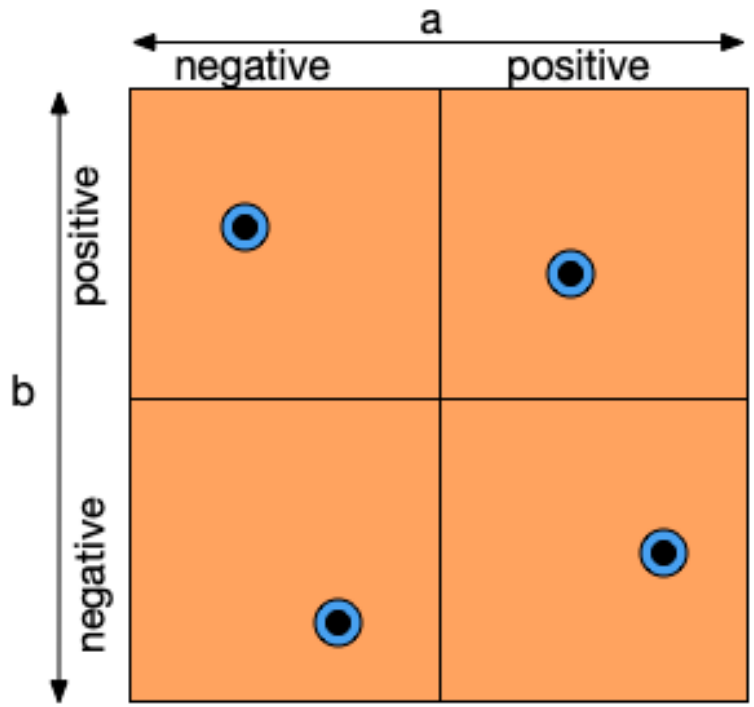
Example : **BigInteger.multiply()**

```
/**
 * @param val another BigIntger
 * @return a BigInteger whose value is (this * val).
 */
public BigInteger multiply(BigInteger val)
```

```
BigInteger a = ...;
BigInteger b = ...;
BigInteger ab = a.multiply(b);
```

multiply : BigInteger × BigInteger → BigInteger

Example : **BigInteger.multiply()**



Que pensez-vous de cette batterie de tests ?

Example : **BigInteger.multiply()**

Toujours inclure les cas limites (frontières des partitions)

a (ou b) est :

- positif
- négatif
- “petit entier” (small integer)
- “grand entier” (big integer)
- égal à zéro
- égal à 1
- égal à -1

On a donc en tout 7x7 = 49 partitions.