Chasse aux bugs:

dysfonctionnement : une erreur, défaut, défaillance, ou panne.

-> défaillance : résultat inattendu, service non rendu
↳ le logiciel fonctionne.

-> panne : suite de défaillance qui cause l'arrêt du prgm

pour éviter tous ça on utilise des méthode de V&V :

-> validation par test.

-> vérification avec des méthodes formels.

⇒ logiciel ce n'est pas seulement du code.

+ des besoin, gestion de projet ... et V&V.

⇝ validation : logiciel répond aux besoins.

⇝ vérification : fonctionne correctement dans n'importe quel situation.

** tester logiciel :

y a 2 types

test structurel.
-> codage est correcte ?
-> lié à l'implémentation.

test non fonctionnel
-> esq l'usage est conforme.
-> lié à la configuration ..

test fonctionnel
-> conforme à la spécification ?
-> test lié à la spécification qualité, performance.

- il y a differents méthodes de tests :
  -> tests en boite blanche (fonctionnel).
    ⇝ sans voir le code, on donne les entrée et les sotie attendue.
  -> tets en boite blanche (structurel).
  :

⇒ que testons nous : les types logiciel.
- transformationnels
- interactif.
:

des exemple de contrainte:
bdd: volume de données, intégrité (bdd est connecté).
web: disponibilité, multi-navigateur ...
compilation: test du langage entrée.

⇝ Principe de test:
- indépendance: un programmeur ne doit pas tester son propre programme.
- Paranoia: pas faire les tests avec l'hypothèse qu'il n'y a pas d'erreur.
- Prédiction: les définition entrée / sortie doit être faite à l'avance.
- vérification: faut inspecter les résultat de chaque test.
- robustesse: les jeux de test doivent être avec des jeux valide et non valide aussi.

→ les étapes du test:

besoins ⟷ definition
spécification ⟷ système
conception ⟷ intégration
code ⟷ unitaire

↑

lors du test on commence par le dernier

⇒ test de validation:
But: est ça répond aux besoins attendue.
quand: ~~quoi~~ dés que l'ensemble des sous systèmes est testé.

Conception des tests
grâce aux spécifications

* Réduire la combinatoire :
  ⇒ All singles :
  * construire des test en fct de la cardinalité de plus grand ensemble (nbr de teste == cardinale + grand).

Rq : si on a le choix on choisi des combinaison logique.

* Avantages :
  ⇝ Réduire nbr de test.
  ⇝ Maintenir une certain qualité de test (chaque paramètre est testé au moins une fois)
  ⇝ détection facile d'une grosse erreur, ou oublie de réalisation
* inconvénients :
  ⇝ difficile de détecter les erreurs lié à une association particulière des paramètre. (lien entre paramètre par exemple).

⇒ All pairs :
  * toutes les pairs de possibilité soient converter au moins par un jeux de tests.
  * ordonner les variables en ordre décroissant en fct de leurs nombres de valeurs.
  * la table soit au moins de taille V1 x V2.

* Avantages :
  → technique très difficile
  → facile à mettre en place.
  → large couverture de cas de tests.
* Inconvenients
  → pour des paramétres de cardinalité très élévé, ne fctionne pas.

⇒ Test par classe d'equivalence : utilisé quand on a des valeurs de grande taille ensemble de valeurs pour lequelle on ne peut distinguer le comportement du logiciel, faut définir les classe valides et invalide.

* Tester aux limites:

tester les valeurs qui sont aux limites (frontières des domaines de fonctionnement logiciels), les frontière se définissent grâce aux domaine de classe d'equivalence (ie donner l'entrée et la sortie du test)

↝ Classe d'equivalence suite:

on peut combiner les classe d'equivalence avec les autres méthodes de tests pour former des jeux de test.

exemple: classe d'equivalence + all single (tester chaque classe d'equivalence au moins une fois que se soit une classe valide ou non valide.

⇒ ainsi qu'on peut utiliser le All pairs:
commencer par la classe qui a le + grand cardinalités et puis alterner les autres valeurs pour avoir des pairs de tous les classes.

↝ lors d'un test de paramètre booléan, faut tester les classes valide ainsi que les classes invalide (faut tester des cas aussi qui se rapproche du valide).

↝ Test grâce aux spécifications:

il existe d'autre ~~the~~ technique de test:
→ Tester grâce à une table de décision.
→ utiliser un diagramme d'état de transition.

Test en boite blanche

* graphe de flot de contrôle : GFC
représenté par :
  - Sommet : les instructions.
  - arc orienté : les branchements.

* on peut distinguer du code mort (instruction qui ne s'exécute jamais) avec le sommet sans arc entrant. (à éliminer cette partie de code)

→ Critère de couverture :

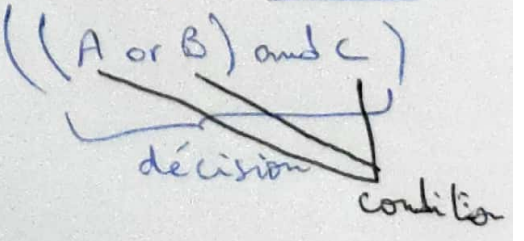
> toute les instruction : tester toute les instruction.
ne permet pas de ...
↳ quand on a seulement un if sans else le critère toute les instruction suffit.

> tous les chemins : les séquences de test doivent couvrir l'ensemble des chemins possible du GFC.
↳ nbr de jeux de test peuvent être infini (combinaison de condition et boucle).

→ toutes les branches et toutes les décisions :
> limitation lors des conditions composé.
> ne permet pas de déconvrir les opérateurs utilisés.

→ toutes les conditions - décisions :
* permet de couvrir toutes les branches et toutes les valeurs de conditions intervenant dans l'expression conditionnelle.
* pour une condition de N valeur faut N+1 tests.
* inconvénients : risque d'avoir des jeux de tests redondants.

→ toutes les conditions - décisions modifiées :
* utiliser dans les système embarqué critique.
* pour une décision contenant N constitions, faut au plus N+1 tests.
* on prend une valeur qui change une condition et influence pour changer la décision.



* Toute conditions - décision modifié :

1- construire une table avec en colonne les conditions et en ligne les différents valeurs des conditions :

⇒ les cas invalide ne traite pas.

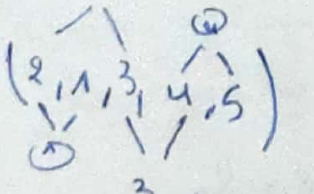
2- choisir les jeux de tests :

→ on compare deux ligne qui ont un seul changeant et qui influence sur la décision.

→ on remplie la colonne de ce changeant avec le numéro de ligne (qui sont complémentaire).

→ on prend les valeurs de deux ligne complémentaire pour chaque condition (on dit que la condition x est couverte par ses 2 lignes).

→ on minimise le nbr de jeux de tests par prendre les ligne qui sont complémentaire et si une ligne de ces dernier est complémentaire avec une autre ligne dans une autre condition on le prend).



à condition décision est changé avec un seul changeant de valeur (complémentaire)

	Con 1	con 2	con 3	con 4	D	Com 1	Com 2	Com 3	Com 4
1	V_1	V_2	V_3	V_4	D_1	2			
2						1	3	4	
3							2	3	
4									5
5									4

* Tous les i - chemins :

→ y a des erreurs qui ne sont pas détecté.

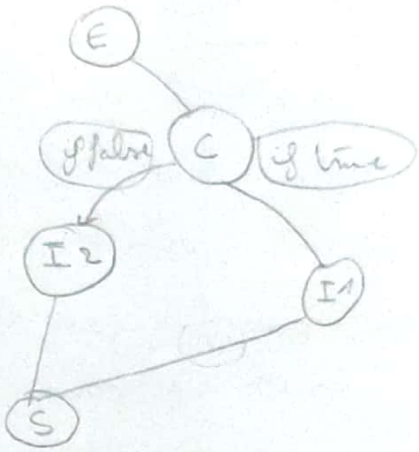
→ Solution : choisir un jeux de test qui passe par la boucle au moins i - fois.

Exercice.

① $a=2, b=1 \Rightarrow (E, C, I1)$

② $(a=1, b=2) \Rightarrow (E, C, I2)$ π^+

③ la division par zéro n'est pas détecter.



→ graph de flot de données: GFD
c'est un GFC mais on ajoutant des info sur les var à tester:

(d): définit

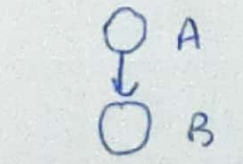
(u): utiliser (dans un calcul ou prédicat).

(cp): → c: utilisé (dans un calcul): donnée
→ p: " (dans un prédicat)

(k): détruit.

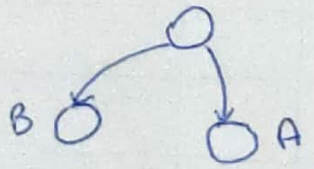
→ utilisation statique de l'équitage des noeuds:
- > un exécution produit un mot sur l'alphabet (u, c, p, d, k)
- > les mots produit un langage, l'analyse du langage revient à analyser les exécutions possible du ~~langage~~ logiciel.

Séquence.



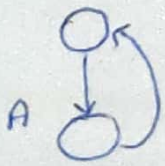
AB

Choix



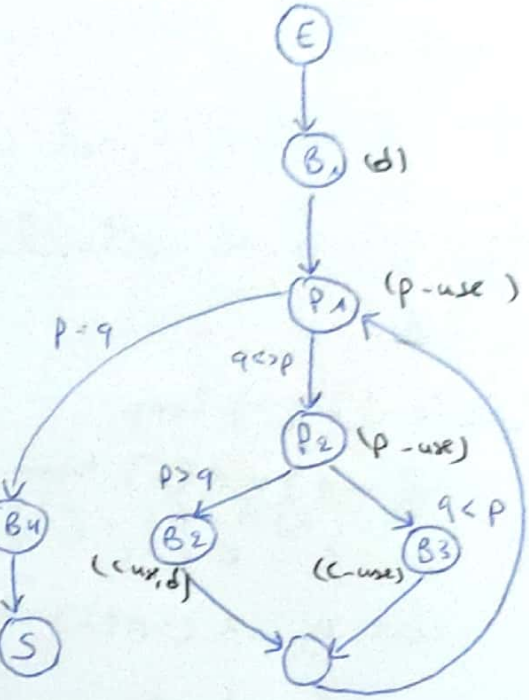
A + B
A ou B

Boucle



A^* ou A^+

Exemple pged:



→ expression régulière :

on parcourt tous les chemins du graphe :

E B1 P1 Pe Be B3 B4 Bu S

K d (P P (cd + c))* P c K

début Boucle — choix — fin Boucle — Boucle s'exécute 0 ou plus fois

⇒ Construction des mots :

en utilisant l'expression régulière :

K d p c K → (E B1 P1 Bu S)

K d p p c d p c K → (E B1 P1 Pe B2 P1 Bu S) ← des chemins.

⇝ il peut y avoir une infinité de mots.

⇒ Critère de couverture : tous les définitions :

> une définition doit être utilisé (suivie par une utilisation) au moins une fois.

⇒ Critère de couverture : tous les utilisations dans un calcul :

> une définition doit être suivi au moins une fois par un c-use.

⇒ critère de couverture : tous les utilisation :

> tous les utilisations doit être couvrir par un jeu de test.

Exercice : cdi :

→ expressions régulière :

• pour "c" :

$K\, d_c\, (c_c\, d_c)^*$

• pour "d" :

$K\, d_d\, (c_d\, d_d)^*$

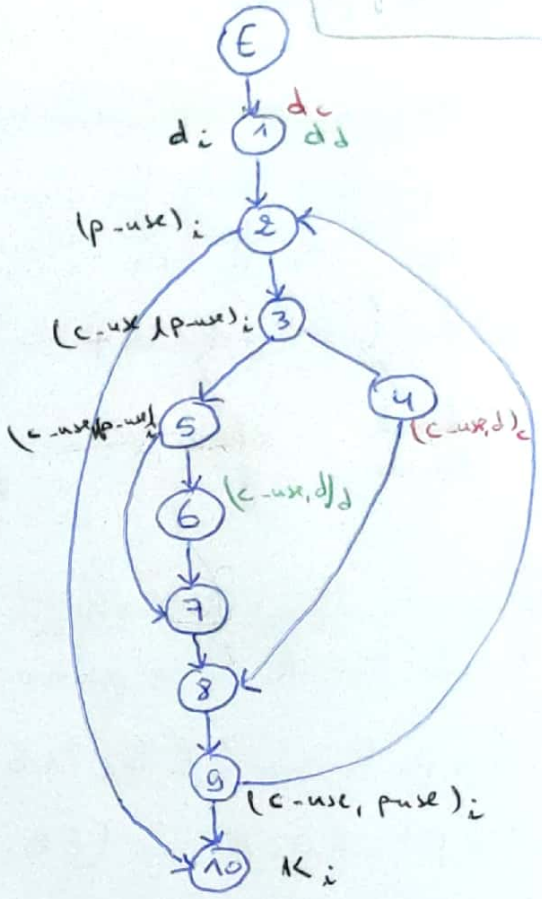
• pour "i" :

~~$K\, d_i\, (P_i^+ [(c_i P_i)^* (c_i P_i)^* (c_i P_i)^*]^* K_i$~~

quand une variable est en output (global) on fait pas le kill(K)

• ~~pour "i"~~ : ~~$K_i\, d_i\, (P_i\, c_i\, P_2\, (c_i P_i)^* (c_i P_i)^*)^* K_i$~~

```
procedure f (....)
 i: natural
Begin
1  c := 0;
   d := 0;
   i := 1;
2  while (i <= n) loop
3     if m1 >= A[i] then
4        c := c + 1;
5     else if m2 <= A[i] then
6        d := d + 1;
7     end if;
8  end if;
9  i := i + 1;
   end loop;
   end f;
```



quand une var est un indice de tab est utilisé 2 fois : calcul et prédicat

*expression régulière pour "i" :

$$di\ P_i \left(C_i\ P_i \left[(C_i\ P_i) + (\ ) \right] C_i\ d_i\ P_i \right)^* K_i$$

(else : $(C_i\ P_i)$; if : $(\ )$)

Exercice 02 : jeux de test "dc i" :

m1 = 12 m2 = 25 n = 1 A[1] = 5 => c = 1, d = 0

E.1.2.3.4.8.9.2.10

1) vérifie le critère de tous les instructions.

2) " " " " les définitions de c

3) vérifie le critère de tous les utilisation de c mais pas pour le d.

m1 = 12 m2 = 25 n = 1 A[1] = 37 => c = 0, d = 1

E . 1 . 2 . 3 . 5 . 6 . 7 . 8 . 9 . 2 . 10 .

1) vérifie le critère de tous les instructions .

2) ne vérifie pas le critère de tous les définition de c .

3) vérifie le critère de tous les utilisation de d mais pas pour le c .

amélioration de jeux de test :

m1 = 12 m2 = 25 n = 2 A[1] = 37 A[2] = 5

Test d'integration

→ on peut pas faire les Test d'intégration sans à l'avance faire les Tests unitaire.

→ test d'integration : Teste le bon fonctionnement de la communication entre les composants.

→ on prepare les tests d'intégration au moment de la conception

→ But : validation des sous-sys
  ↳ Test d'intégration Logiciel / Logiciel.
  ↳ " " Logiciel / Materiél.

→ Type de test : des interfaces (car on teste la communication et cette dernière se fait via des interfaces.

→ on peut tester les interation entre classe (module).

→ identifier les dépendance entre " "

→ modéliser " " chaque classe et son environt.

→ choisir un ordre pour l'intégration.

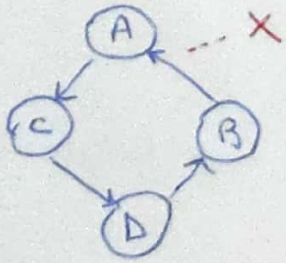
⇒ graphe acyclique :

à partir d'un modèle de conception en sortie un méta modèle modélisant seulement les modules (noeud) et les dépendance entre module (relation entre module).

acyclique : car on veut pas tomber sur des dépendance forte

→ l'ordre des tests d'integration :

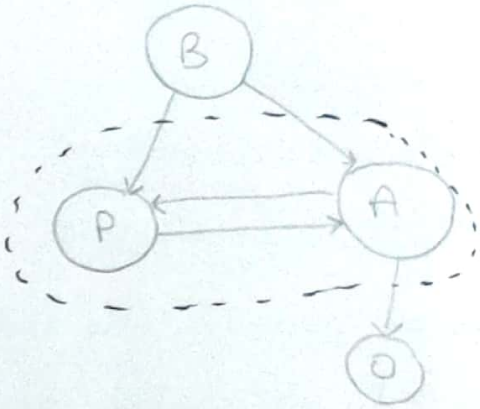
on commence par les noeuds qui sont indépendants



les tests se fait séquentiellement.

Exemple
graphe cyclique

→ pour régler ça il faut casser les cycle



→ implémenter une simulation concrète c'est le Stub.

par exemple simuler le comportement d'une méthode

→ possède l'interface de la classe simuler avec un comportement contrôlé (sortie statique)

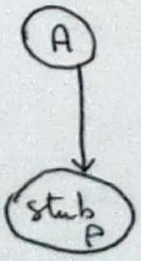
→ le stub simule les sortie d'un module et les d'autres entré.

* déterminer les méthodes de P utilisé par A
* définir le stub lié à la classe P

→ quand on teste A l'autre classe on la rend sous forme de stub (cas de P).

→ les méthodes lié à d'autre classe on peut pas les testé dans les tests unitaire donc on est obligé de les laisser jusqu'aux tests d'integration.

~> tester la classe A avec le stub de P



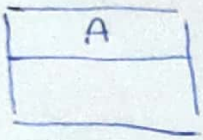
~> tester la classe P avec la classe A

↝ tester la classe A avec la vrai classe P.

⇒ graphe de dépendances de tests :

- Type de noeud :

→ classe :

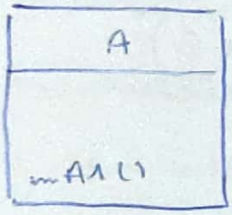


classe A

⇒

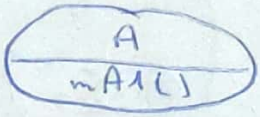
noeud A

→ méthodes :



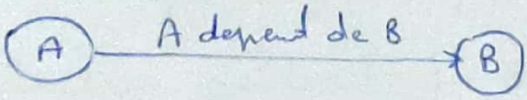
méthode de classe A

⇒



- Type d'arc :

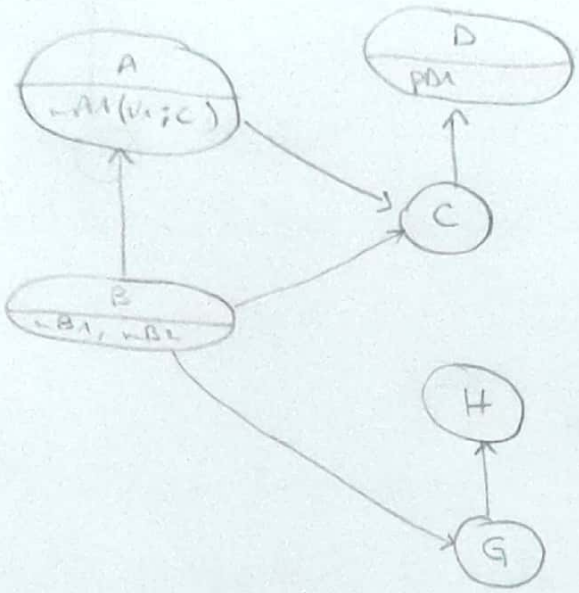
→ classe à classe :

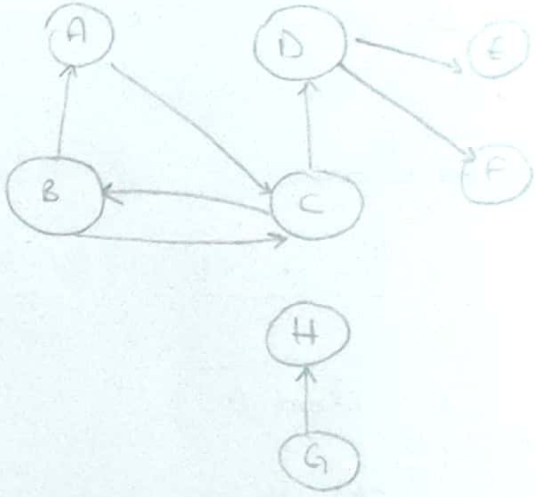


→ méthode à classe :

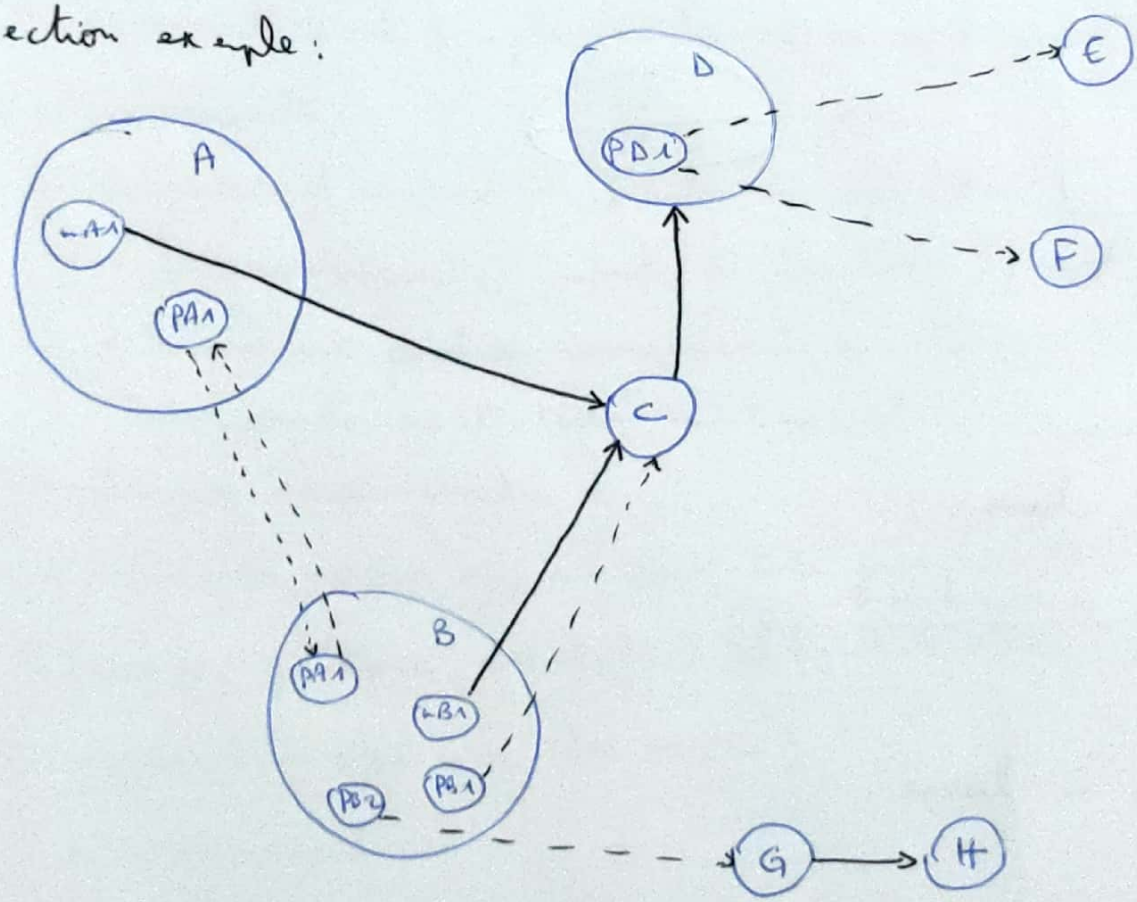
→ méthode à méthode :

exemple :





connection exemple :



----> relation des méthode privé.

Stratégie d'integration :

* Types de Stub :

→ stub réaliste : simule tous les comportements.

→ " spécifique : " le comportement d'un seul clients.
(stub c' pour A et stub c" pour B).

⇝ Stratégie big-bang :

tous les composants à testé sont integré en une seule étape (si les composants sont simple facile et on voit que c'est faisable).

* inconvénients :
  - cause d'erreurs difficile à détecter.
  - la complexité induit à des tests manquants.
  - testé d'integration commencent que lorsque tous les composants sont testé unitairement.

⇝ Stratégie descendante :

* on integre nv par nv, on doit testé par les stub (car y a des dependance)
* détection précoce des défauts d'architecture (ça va corriger la conception s'il y a des erreur).

* inconvénient :
  - pluriens stubs à mettre en place.
  - multiplie le risque d'erreur.

⇝ Stratégie ascendante :

- testé du nv par au nv Haut.
- pas de stub à mettre en place.
- jeux de tests définit facilement.

⇝ Stratégie sandwich :

- Choisir une couche simple (testé les nv haut et les nv bas jusqu'à l'arrivé au nv le plus complexe au milieu) puis testé avec le reste, le tous ensemble.
- pour ne pas perdre de temps à attendre une couche qui peut être complexe en implementation ou test.

⇒ Avantage :

- premier composants terminés sont intégré en premier.

⇒ inconvénient :

- ne teste pas tous les composants de la couche cible.

Mock et Stub

Un mock est un type de stub, il simule le comportement d'1 objet distant (objet réel exterieur ex: BDD, web service).
mock permet de tester.

→ une classe en isolation
→ les interactions avec les environnement.

* mettre en place un mock:

→ créer un mock
→ définir les comportements du mock.
→ écrir et exécuter les tests.
→ vérifier le comportement du mock.

* définition de mock:

```
personne mockp = mock(personne.classe)  → exemple.
```

ou

```
@Mock.
personne mockp;
```

* fixer la valeur des retour:

```
when (mockp.operation()).thenReturn(SomeResult);
```

* Rétablir le comportement de la méthode:

```
when (mockp.operation()).thenCallRealMethod();
```

* changer le comportement d'une méthode void:

```
when (mockp.operation(Mockito.anyString())).DoNothing();
```

* Lancer une exception:

```
when (mockp.operation()).thenThrow(new IllegalArgumentExcepting);
```

* vérifier le comportant:

```
verify (mockp).methode(valeurArgument);
```

* vérifier l'ordre des invocation d'objet :

→ InOrder.verify(mockp);

⇒ annotation @Spy :

↳ le premier que je les écrit veut dire que c'est le premier exple :

InOrder.verify(mock1); ← 1er
" .verify(mock2); ← 2ème
} dans l'ordre.

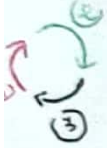
Les limites de mockito :

* impossible de mocker une classe ou interface final (constate).
* " " une méthode statique ou privé.
* " " une méthode equals() et hashcode()
(mockito dépend sur ces 2) → (il faut toujours redéfinir ces 2 méthodes).

Dév dirigé vers les tests

TDD : Test dirigé dev

- généralement intégré aux approach de dev agile.
- utilisation des tests unitaire comme spécification du code (on commence par les test puis le developpement).
  * le cycle rouge - vert - gris (refactor)



rouge : on écrit des test (qui ne fonctionne pas)

vert : . . . . . le code qui fait fonctionner les test

gris (refactor) : ré-effectuer le travail afin d'améliorer le code.

→ code le plus simple possible juste pour faire passer les tests

Avantages TDD :

* Raisonner sur les spéc avant d'écrire le code.
* Construire le code étape par étape (spirale).
* Méthode plus productive (gain en temps et en qualité).
* Détecter les erreurs le plus tôt possible.
* Se concentrer sur une fonctionnalité à la fois.