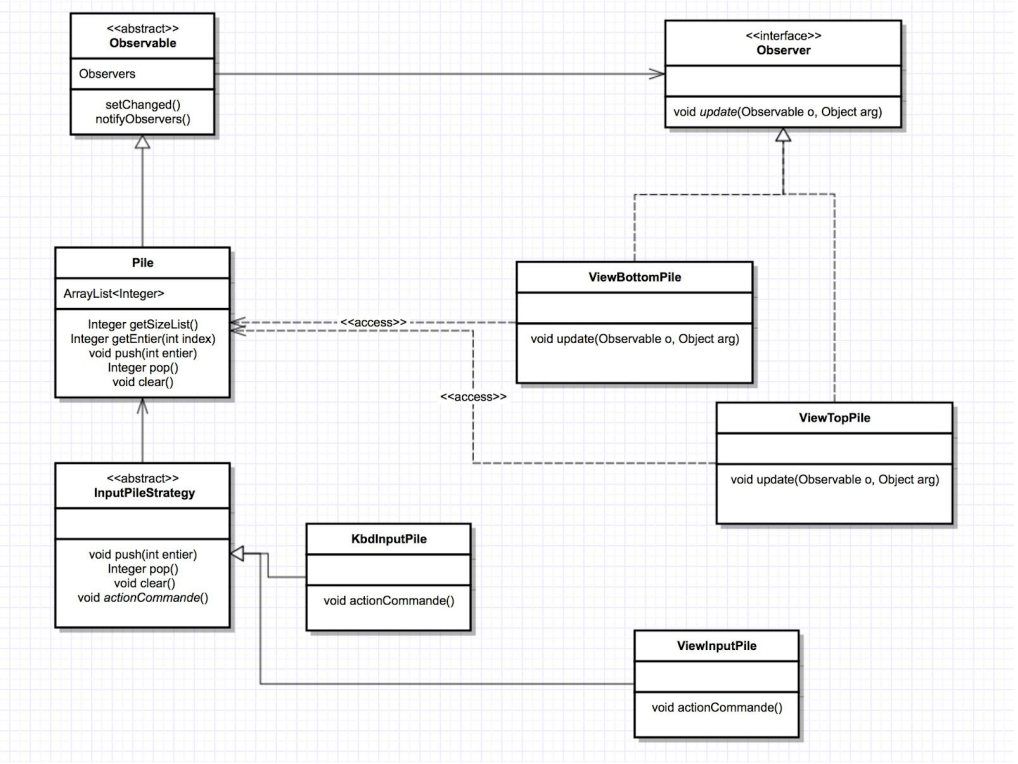
Rapport de conduit de test

ABULIMITI Alafate
LI Yuanyuan

Table de matière

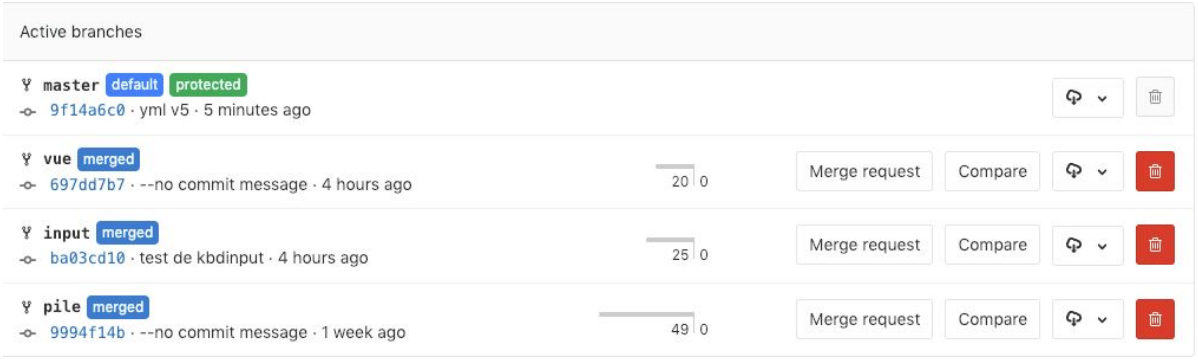
1. Première étape (1ère séance)

Dans notre projet, nous avons développé un programme selon le diagramme de classe.

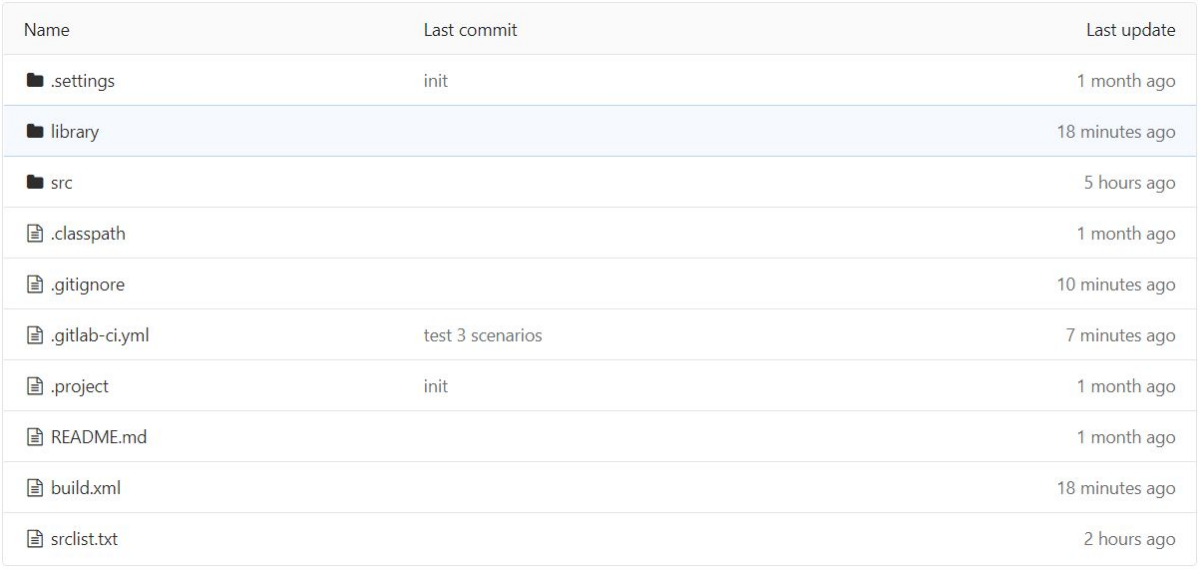


Le modèle d'observateur utilisé par ce projet peut être trouvé dans la figure.
Le mode Observateur sert à établir une dépendance entre un objet et un objet. Lorsqu'un objet change, il en informe automatiquement les autres objets, qui répondent en conséquence. En mode observateur, l'objet qui change est appelé cible d'observation et l'objet notifié est appelé observateur.Une cible d'observation peut correspondre à plusieurs observateurs et il n'y a pas de connexion mutuelle entre ces observateurs. Ajouter et supprimer des observateurs pour faciliter l'échelle du système. Un système logiciel nécessite souvent qu'un autre objet apporte un changement correspondant lorsque l'état d'un objet change. Il existe de nombreuses solutions de conception pour ce faire, mais pour faciliter la réutilisation du système, vous devez choisir une conception à couplage faible. La réduction du couplage entre objets facilite la réutilisation du système, mais le concepteur doit également maintenir la coordination des actions entre ces objets à faible couplage, en assurant un degré élevé de collaboration. Le modèle Observer est la plus importante des différentes options de conception répondant à cette exigence.

Nous avons construit notre projet dans gitlab selon les besoins et créé trois branches.

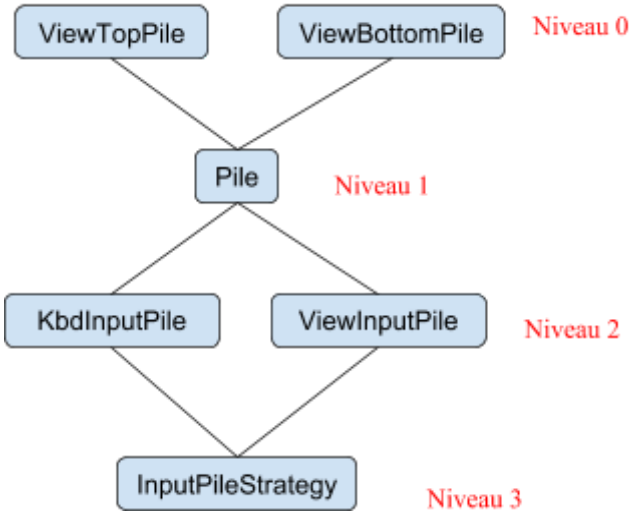


Le site Web est comme suit : https://gitlab.com/felixneoaa/conduitedetest



Ici, nous utilisons Junit comme bibliothèque de test unitaire et nous prévoyons d'utiliser CI comme intégration de test.

L'architecture de classes est comme ci-dessous:



L'architectures de classes (L'arbre de classes)

Il y a quatre niveaux:

- Niveau 0: la classe ‘InputPileStrategy’. c’est la base de reçoit les requêtes de client.
- Niveau 1: ce sont les classes qui hérite de la classe ‘InputPileStrategy’.
- Niveau 2: la classe ‘Pile’ qui est plus immportant de traiter la pile.
- Niveau 3: ce sont les classes envoient le résultat de traitement de la pile.

On divise ces quatre niveau a deux grand niveau: N1 (n0+n1) et N2 (n2+n3). Et pour chaque grand niveau on utilise le modèle ‘**Bottom-up**’ pour développer.

Donc, notre plan est comme ci-dessous:

Etape 1:

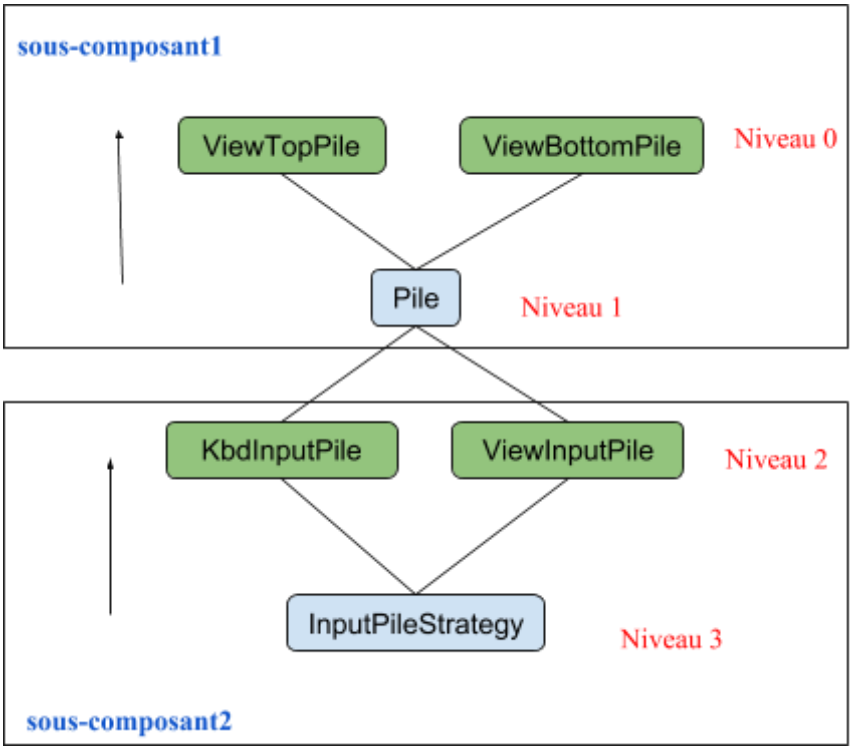
- Développer la classe ‘InputPileStrategy’ et la classe ‘Pile’.
- Tester.

Etape 2:

- Développer la classe ViewTopPile , ViewBottomPile.
- Intégrer ces deux classes avec la classe Pile. (**sous-composant1**=intégrer niveau0 et niveau1)
- Tester.

Etape 3:

- Développer la classe - KbdInputPile, ViewInputPile. (**sous-composant2**=integrer niveau2 et niveau3)
- Intégrer le composant complet (intégrer sous-composant 1 et sous-composant 2).
- Tester.



‘Bottom-up’ de sous-composant

Après le premiere etape, on juste programme la classe ‘Pile’ et la classe ‘InputPileStrategy’. Donc, quand on fait des testes on a besoin de pilote ou bouchon.

Exemple de Pilote : ViewTopPile (pas de classe vraie)

```
public class ViewTopPile implements Observer{

    public  int count=0;

    @Override
    public void update(Observable o, Object arg) {
        // TODO Auto-generated method stub
        Pile pile =(Pile) o;
        count++;
        System.out.println("pilote viewTop");
    }
}
```

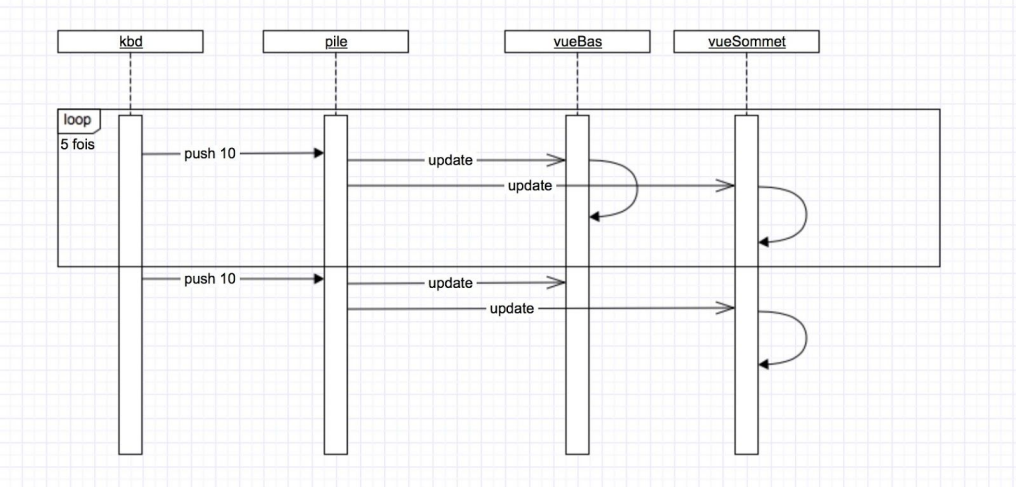
Pilote : ViewTopPile

Exemple de Bouchon: viewInputPile(pas de classe vraie)

```
public class ViewInputPile extends InputPileStrategy {
    public String action;
    public int num = 0;

    @Override
    public void actionCommande() {
        if(action.equals("pop"))
            this.pop();
        else if(action.equals("push"))
            this.push(num);
        else if(action.equals("clear"))
            this.clear();
    }
}
```

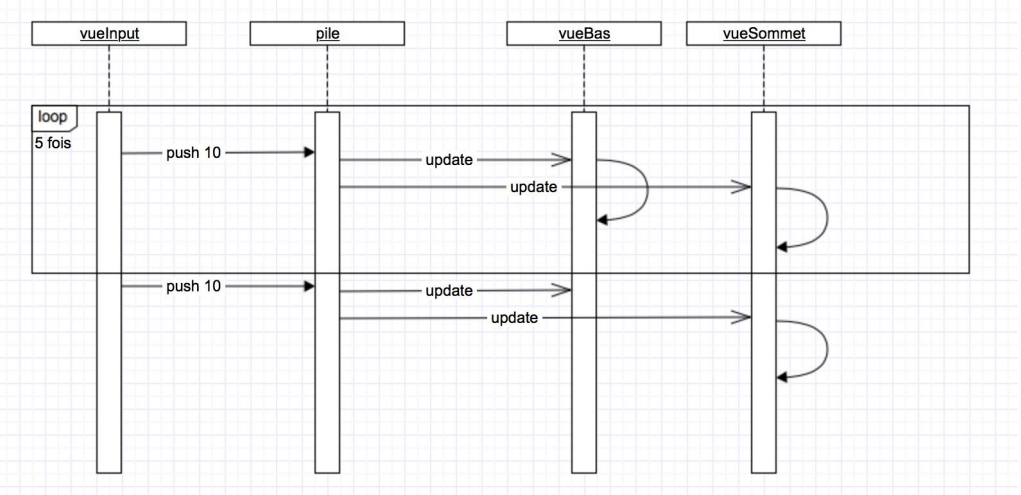
Bouchon: viewInputPile

test*Scénario1*: bouchon(KbdInputPile) + Pile + Pilote(ViewTopPile, ViewBottomPile).



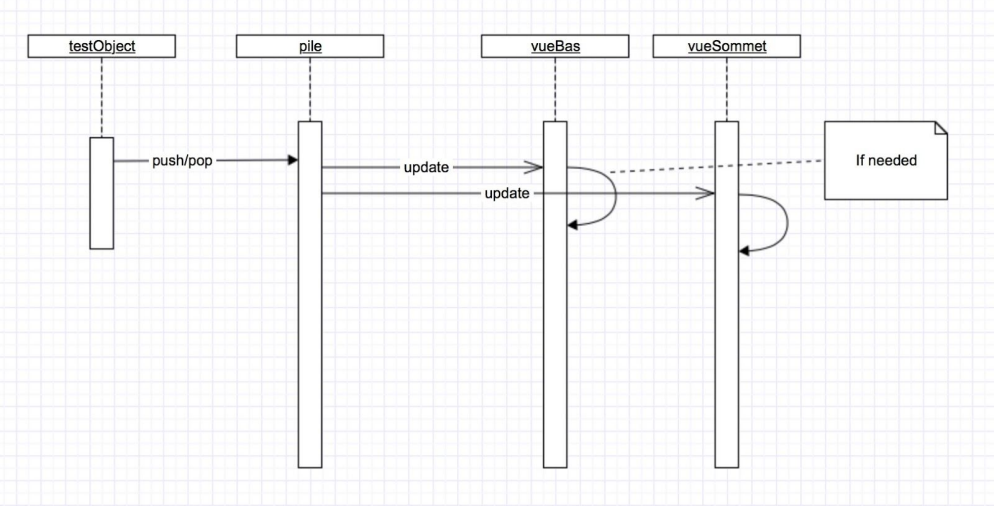
Scénario1

test*Scénario2*: bouchon(viewInputPile) + Pile + Pilote(ViewTopPile, ViewBottomPile).



Scénario2

test*Scénario3*: Pile + Pilote(ViewTopPile, ViewBottomPile).



Scénario3

2. Deuxième étape (1ère et 2ème séance)

Le fichier TestPile.java est un fichier de test qui spécialise les tests sur la classe Pile:

```java
public class TestPile {

    private Pile pile = new Pile();

    @Test
    public void testPush() {
        pile.clear();
        pile.push(5);
        assertEquals(5,pile.getEntier(0).intValue());
    }

    @Test
    public void testPop() {
        pile.clear();
        pile.push(5);
        int pop=pile.pop().intValue();
        assertEquals(5,pop);
    }

    @Test
    public void testSize() {
        pile.clear();
        pile.push(1);
        pile.push(2);
        pile.push(3);
        pile.push(4);
        pile.push(5);
        assertEquals(5,pile.getSizeList().intValue());
    }
}
```

Nous avons utilisé la bibliothèque JUnit 4 et, pour chaque méthode de la classe Pile, nous avons implémenté la méthode de test correspondante. La déclaration de la méthode de test utilise les commentaires fournis par la bibliothèque. De plus, afin de vérifier les résultats, nous avons utilisé la méthode assertEquals (), qui nous indique l'état du test. En fait, les objectifs de ces trois méthodes sont très clairs, ils peuvent être simulés, tels que pousser la pile, vider et vider la pile.
Nous avons développé et testé trois parties séparément au besoin, et si nous avons trouvé des problèmes sur d'autres branches dans le processus, nous pourrions faire des changements.

Selon les trois scénarios de test précédents,. Nous avons fait les tests correspondants. Pour le premier scénario de test:

```java
public class TestScenario1 {

    @Test
    public void pileScenario1() {
        KbdInputPile kbdInputPile=new KbdInputPile();
        ViewBottomPile viewBottomPile=new ViewBottomPile();
        ViewTopPile viewTopPile=new ViewTopPile();
        kbdInputPile.pile.addObserver(viewBottomPile);
        kbdInputPile.pile.addObserver(viewTopPile);
        kbdInputPile.action="push";
        kbdInputPile.num=10;
        for(int i=0;i<5;i++) {
        kbdInputPile.actionCommande();
        }
        assertEquals(10,viewTopPile.getTop());
        assertEquals(10,viewBottomPile.getBottom().get(0).intValue());
        kbdInputPile.num=11;
        kbdInputPile.actionCommande();
        assertEquals(11,viewTopPile.getTop());
        assertEquals(10,viewBottomPile.getBottom().get(0).intValue());
    }

}
```

Pour le deuxième scénario de test:

```java
public class TestScenario2 {

    @Test
    public void vueScenario2() {
        ViewInputPile viewInputPile=new ViewInputPile();
        ViewBottomPile viewBottomPile=new ViewBottomPile();
        ViewTopPile viewTopPile=new ViewTopPile();
        viewInputPile.pile.addObserver(viewBottomPile);
        viewInputPile.pile.addObserver(viewTopPile);
        viewInputPile.action="push";
        viewInputPile.num=10;
        for(int i=0;i<5;i++) {
            viewInputPile.actionCommande();
        }
        assertEquals(10,viewTopPile.getTop());
        assertEquals(10,viewBottomPile.getBottom().get(0).intValue());
        viewInputPile.num=11;
        viewInputPile.actionCommande();
        assertEquals(11,viewTopPile.getTop());
        assertEquals(10,viewBottomPile.getBottom().get(0).intValue());
    }

}
```

Pour le troisème scénario de test:

```java
*/
public class TestScenario3 {
    @Test
    public void pileScenario3() {
        Pile pile=new Pile();
        ViewBottomPile viewBottomPile=new ViewBottomPile();
        ViewTopPile viewTopPile=new ViewTopPile();
        pile.addObserver(viewTopPile);
        pile.addObserver(viewBottomPile);
        pile.push(1);
        assertEquals(1,viewBottomPile.getBottom().get(0).intValue());
        assertEquals(1,viewTopPile.getTop());
        pile.push(1);
        pile.push(2);
        pile.push(3);
        pile.push(4);
        assertEquals(1,viewBottomPile.getBottom().get(0).intValue());
        assertEquals(4,viewTopPile.getTop());
        pile.push(5);
        assertEquals(1,viewBottomPile.getBottom().get(0).intValue());
        assertEquals(5,viewTopPile.getTop());
        int pop=pile.pop();
        assertEquals(1,viewBottomPile.getBottom().get(0).intValue());
        assertEquals(4,viewTopPile.getTop());
        pile.clear();
        assertEquals(0, viewBottomPile.getBottom().size());
        assertEquals(-1, viewTopPile.getTop());
    }

}
```

Ces tests ont été menés entièrement conformément aux trois scénarios de test requis.

3. Etapes suivantes (CI)

3.1. Configurer CI

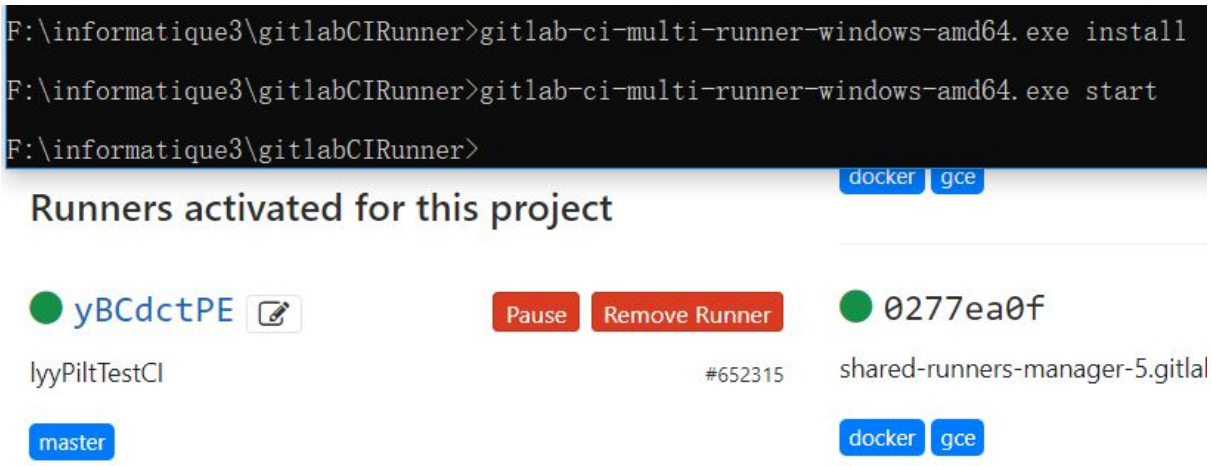
Pour créer et configurer CI, on fait les étapes suivantes:
(1). Enregistrer ‘Runner’:
Il faut créer ‘Runner’ et connecter avec le projet par utiliser ‘URL’ et ‘Token’ de projet.

```
F:\informatique3\gitlabCIRunner>gitlab-ci-multi-runner-windows-amd64.exe register
Please enter the gitlab-ci coordinator URL (e.g. https://gitlab.com/):
https://gitlab.com/
Please enter the gitlab-ci token for this runner:
rFE6haJuC-vCivgkZcv3
Please enter the gitlab-ci description for this runner:
[LiYuanyuan]: lyyPiltTestCI
Please enter the gitlab-ci tags for this runner (comma separated):
master
Whether to run untagged builds [true/false]:
[false]:
Whether to lock Runner to current project [true/false]:
[false]:
Registering runner... succeeded                     □[0;m  runner□[0;m=rFE6haJu
Please enter the executor: parallels, shell, docker-ssh+machine, docker, docker-ssh, ssh, virtualbox, docker+machine, kubernetes:
shell
Runner registered successfully. Feel free to start it, but if it's running already the config should be automatically reloaded!□[0;m
```

Configuration de ‘Runner’

Quand on met à jour le web site, on peut trouver notre ‘Runner’ est déjà dans la liste de ‘Runners’. (avec un point d'exclamation devant ID de ‘Runner’)

(2). Après, il faut installer et démarrer ‘Runner’. si il y a un cercle vert devant ID de ‘Runner’, c’est à dire que notre ‘Runner’ est créée bien.



Demarrer ‘Runner’

(3). Quand on démarre ‘Runner’, on peut vérifier le statut de ‘Runner’ vers le management de service du système comme l’image ci-dessous.

Verification le status de ‘Runner’

(4). Aussi, on peut vérifier les informations dans le fichier ‘config.toml’ comme l’image ci-dessous.

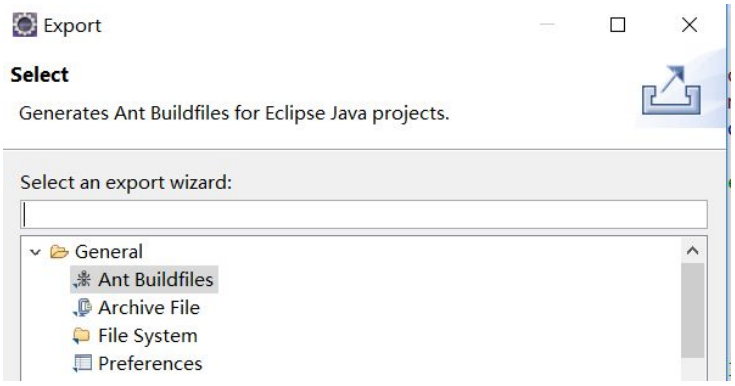
```
[[runners]]
  name = "lyyPiltTestCI"
  url = "https://gitlab.com/"
  token = "yBCdctPEjXyv99NZEojf"
  executor = "shell"
  [runners.cache]
```

Information du fichier ‘config.toml’

3.2. L’integration avec CI

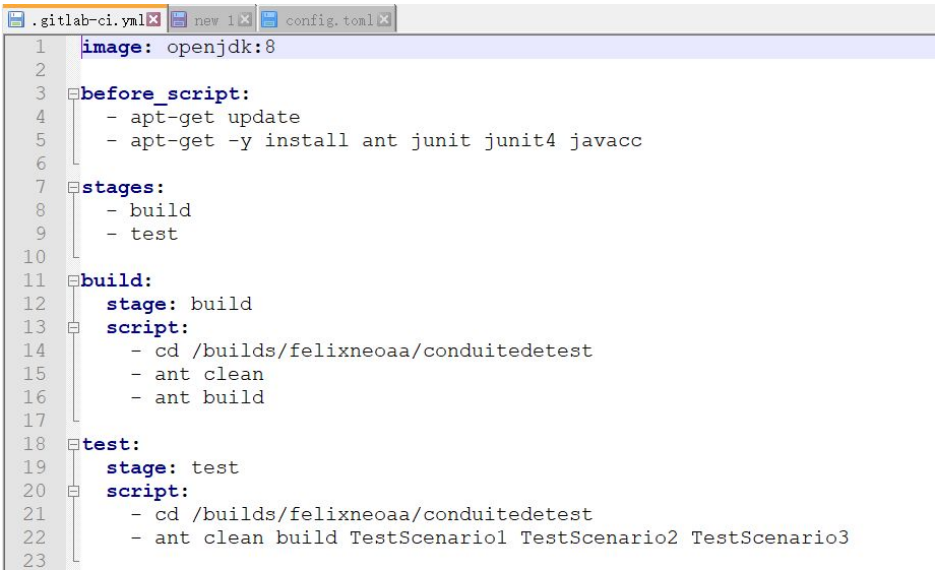
3.2.1. Le fichier ‘.gitlab-ci.yml’

Pour faire l'intégration avec CI, il faut écrire le fichier ‘.gitlab-ci.yml’ pour configurer CI.
D’abord, on utilise ‘Eclipse’ pour générer le fichier ‘build.xml’ qui sauvegarde tous les informations de configurations du projet.



Générer le fichier ‘build.xml’

Apres, notre fichier ‘.gitlab-ci.yml’ est comme l’image ci-dessous. il contient deux processus principaux. un est ‘build’ qui permet de compiler notre projet, l’autre est ‘test’ qui est pour faire des testes.



Le fichier ‘.gitlab-ci.yml’

Il faut bien configurer le fichier ‘.gitlab-ci.yml’. Si non, il ne peut pas compiler le projet sur ‘CI’.
Quand on fait le fichier ‘.gitlab-ci.yml’, on se rencontre des problèmes. par exemple, il ne peut pas trouver le librairie, donc il ne peut pas compiler.

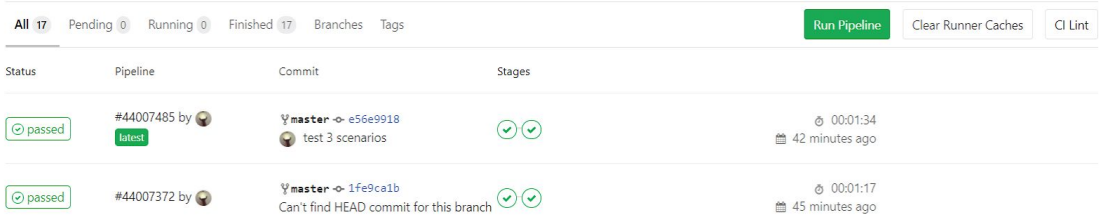
```
build-project:
     [echo] tpPile: /builds/felixneoaa/conduitedetest/build.xml
    [javac] Compiling 17 source files to /builds/felixneoaa/conduitedetest/bin
    [javac] /builds/felixneoaa/conduitedetest/src/test/TestPile.java:3: error: package org.junit does not exist
```

Le probleme

En conséquence, pour résoudre ce problème, on crée un dossier qui contient tous les fichiers de librairie. on prend beaucoup de temps pour résoudre ce problème.

3.2.2. Iteration de l'integration

Chaque fois quand on faire 'push', 'CI Runner' démarre, compile et fait de tests automatiquement. Par exemple, comme notre fichier '.gitlab-ci.yml', on fait test avec TestScenario1, TestScenario2, TestScenario3.

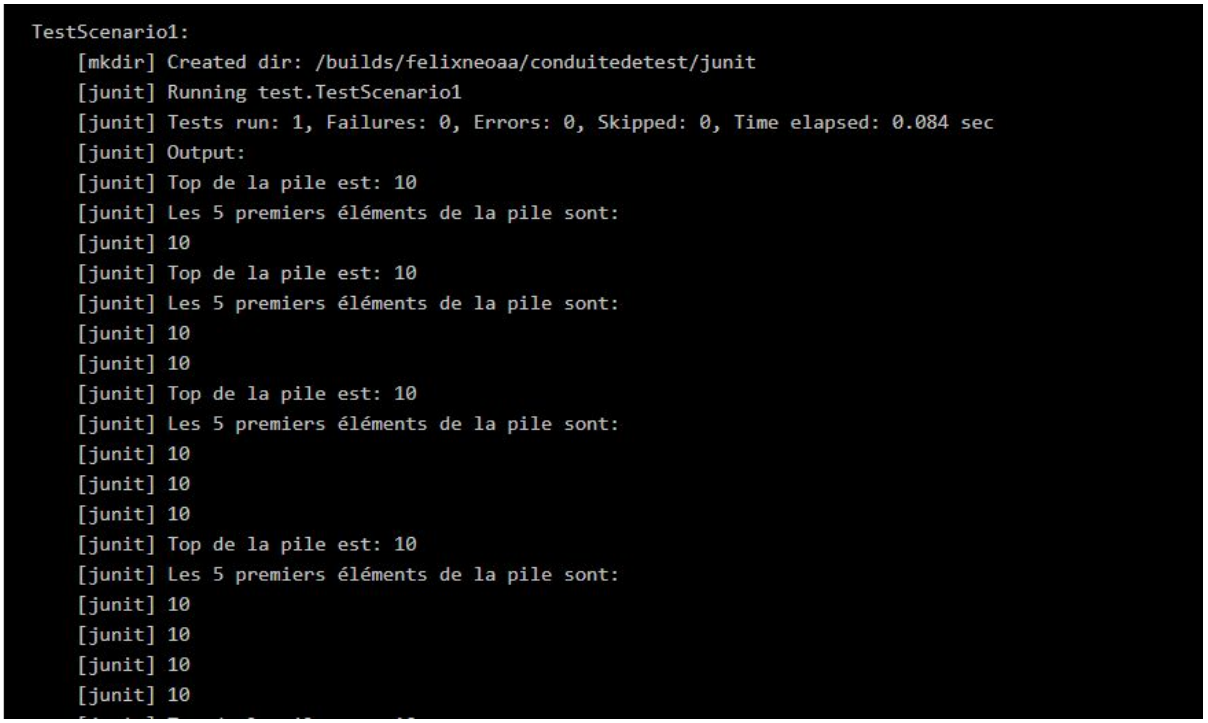


Pipeline

Le résultat de 'jobs' est comme l'image ci-dessous. on peut savoir le résultat de chaque test.

```
BUILD SUCCESSFUL
Total time: 0 seconds
$ ant build
Buildfile: /builds/felixneoaa/conduitedetest/build.xml

build-subprojects:

init:
    [mkdir] Created dir: /builds/felixneoaa/conduitedetest/bin

build-project:
     [echo] tpPile: /builds/felixneoaa/conduitedetest/build.xml
    [javac] Compiling 17 source files to /builds/felixneoaa/conduitedetest/bin

build:

BUILD SUCCESSFUL
Total time: 1 second
Job succeeded
```

Le resultat de 'job'- build



Le resultat de 'job'- test

Donc, chaque fois, quand on fait 'push', cela permet de compiler et tester automatiquement sur CI de 'Gitlab'.

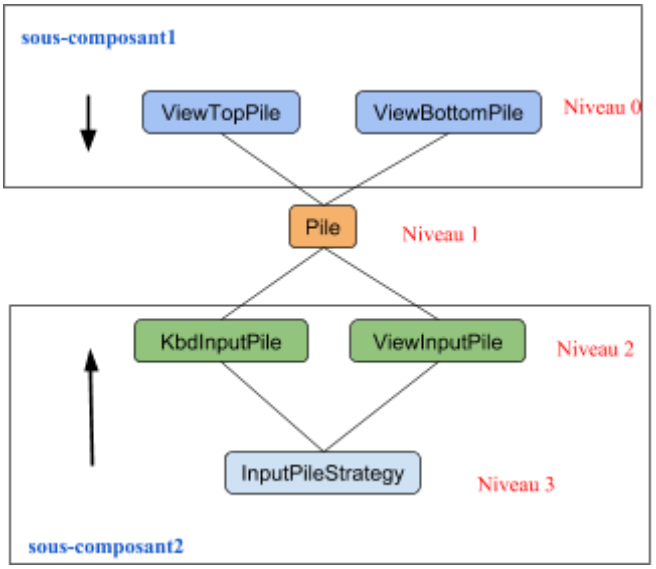
4. Conclusion

Vers ce tp, on a fait l'intégration de java Application qui permet de faire 'push', 'pop' et 'clear' dans une pile. En plus, chaque fois, quand la pile met à jour, il affiche les premiers éléments de la pile et l'élément de 'Top' de la pile.

Séance	**Evennements**
Séance1	- Développer la classe 'InputPileStrategy' et la classe 'Pile'. - Tester.
Séance2	- Développer la classe ViewTopPile , ViewBottomPile. - Intégrer ces deux classes avec la classe Pile. - Tester.
Séance3	- Développer la classe - KbdInputPile, ViewInputPile - Intégrer le composant complet.
Hors de Séance	- Créer CI Runner. - Démarrer, tester et intégrer.

Tableau de réalisations de Séance

Après notre réalisation, on pense qu'on peut essayer d'utiliser le modèle **‘Sandwich’** . Comme l’image ci-dessous.



- Développer la classe la classe ViewTopPile , ViewBottomPile.
- Tester: (ViewTopPile ,ViewBottomPile) + Bouchon (Pile).
- Développer la classe la classe InputPileStrategy, KbdInputPile, ViewInputPile.
- Tester: (KbdInputPile, ViewInputPile) + Pilote (Pile).
- Développer la classe Pile, integrer composant complet et tester.