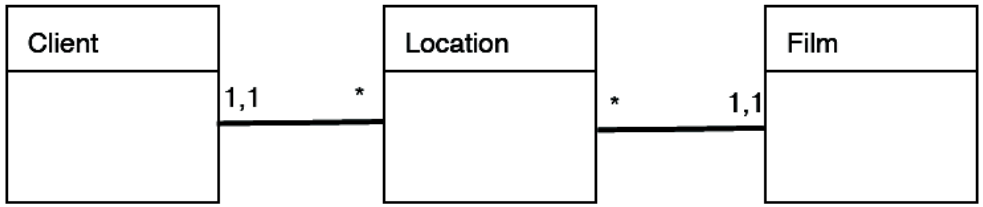
Test en isolation

Indépendance des tests

Le résultat d'un test ne doit pas dépendre de l'exécution des tests précédents pour s'assurer que l'échec d'un test n'est pas lié à une de ses dépendances.




```
public class Location {
 private Film film;
 private Client client;
 ...
 public float montant (int duree)
   { if (client.getCat() ==
        PRIVILEGE)
       return film.prixJour()*(
           duree-1);
    else ...
 }}
```

```
public class Film {
 private Categorie categorie;
 private String titre;
 ...
 public float prixJour(){
    switch(categorie){
     case Categorie.NOUVEAUTE:
      return categorie.prixBase()
           *...:
      ...
 }}
```

Si erreurs dans classe `Film`, tests sur méthode `montant` peuvent échouer à cause d'erreurs dans la méthode `prixJour`.

Mise en oeuvre du TU automatique

Mise en oeuvre de tests en isolation

Le résultat d'un test ne doit pas dépendre de l'exécution des tests précédents :

- utilisation de design patterns
- utilisation d'objets de type *mock* ou doublures d'objets
- éviter de faire appels aux ressources externes dans les cas de tests
- ...

Les doublures d'objets

Les doublures d'objets permettant de simuler le comportement d'autres objets de façon maîtrisée :

- *stub* (bouchon) : classes qui renvoient en dur une valeur pour une méthode invoquée
- *spy* (espion) : classe qui vérifie l'utilisation qui en est faite après l'exécution
- *mock* (simulacre) : classe qui agit comme un *stub* et un *spy*

Doublures d'objets

Intérêts des Mocks

- Utiles pour les TU, pour simuler un objet qui n'est pas encore écrit, pour éviter d'invoquer des ressources longues à répondre, pour obtenir un état difficilement reproductible
- Maitrise des dépendances durant un test
- Réaliser les tests d'un objet de façon isolée et répétable.
- Vérifier les invocations qui sont faites sur un objet (nombres d'invocations, paramètres fournis, ordre d'invocations, ...)

Mise en oeuvre des Mocks

- Création dynamique d'objets mocks, généralement à partir d'interfaces.
- Mockito `http ://code.google.com/p/mockito/`
- JMock, EasyMock, ...

Mockito

Générateur automatique de doublures :

1. Création des mocks pendant la phase de création des objets du test
2. Description du comportement des objets (bouchonnage - stubbing) pendant la phase de création des objets du test
3. Lors de l'exécution du code à tester, mémorisation des interactions avec les mocks (observateur - vérification)
4. L'oracle peut interroger les mocks pour savoir comment ils ont été utilisés

Mockito

1- Import et déclaration de Mockito comme une extension à notre classe de tests :

- `import static org.mockito.Mockito.*;`
- `@ExtendWith(MockitoExtension.class)`
  `public class TestLocation`

2- Déclaration des mocks avec l'annotation @Mock :

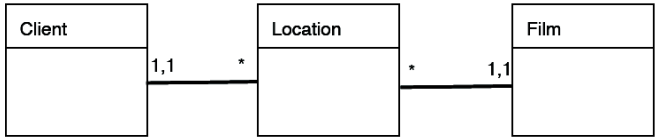
- `@Mock`
  `Film mockfilm`
- Le mock peut appeler tous les appels de méthode de l'interface/classe `Film`.

Mockito

3- Description du comportement des objets (bouchonnage - stubbing) (GIVEN) :

- Retour d'une valeur :
  `when((mockfilm.prixJour()).thenReturn(3.5) ;`
- Levée d'exception : `doThrow(new MonException()).when(loc).montant(10);`
- Stubbing avancé :
  `when((mockfilm.prixJour()).thenReturn(3.5).thenReturn(3)`
- `when(loc.montant(10)).thenThrow(new MonException())`

Mockito




```
public class Location {
 private Film film;
 private Client client;
 ...
 public float montant (int duree)
   { if (client.getCat() ==
        PRIVILEGE)
       return film.prixJour()*(
           duree-1);
    else ...
 }}
```

```
public class Film {
 private Categorie categorie;
 private String titre;
 ...
 public float prixJour(){
    switch(categorie){
     case Categorie.NOUVEAUTE:
      return categorie.prixBase()
           *...:
      ...
 }}
```

```
 @Mock Film mockfilm;
 @Mock Client mockclient;

@Test
public void testMontant() {
 when(mockfilm.prixJour()).thenReturn(3.5) ;
 when(mockclient.getCat()).thenReturn(PRIVILEGE) ;
 Location loc = new Location(mockfilm, mockclient) ;
 Assert.assertEquals(3.5, loc.montant(2)) ;
}
```

Mockito

4- Lors de l'exécution du code à tester, mémorisation des interactions avec les mocks (observateur)

- L'oracle peut interroger les mocks pour savoir comment ils ont été utilisés
- par ex. avec la méthode `verify`

```
 @Mock Film mockfilm;
 @Mock Client mockclient;

@Test
public void testMontant() {
 when(mockfilm.prixJour()).thenReturn(3.5) ;
 when(mockclient.getCat()).thenReturn(PRIVILEGE) ;
 Location loc = new Location(mockfilm, mockclient) ;
 loc.montant(2);
 verify(mockfilm, times(1)).prixJour();
}
```

Mockito

Argument matchers

- permettent de ne pas préciser une valeur d'argument
- `anyInt()` tout entier, Integer ou null
- `anyString()`
- `any(GregorianCalendar.class)`
- ...

```
verify(loc, never()).montant(anyInt())
doThrow(new MonException()).when(loc).montant(anyInt());
```