Génie Logiciel Avancé : dépendances et tests

Arnaud Labourel (arnaud.labourel@univ-amu.fr)

29 janvier 2025

amU Faculté des sciences Aix Marseille Université

Section 1

Tests et dépendances

Tests

Règle

Un code non testé n'a aucune valeur $\Rightarrow$ tout code doit être testé

Différents types de tests

- **Tests unitaires** : Tester les différentes parties (méthodes, classes) d'un programme indépendamment les unes des autres.
- **Tests d'intégration** : Tester une portion du programme (plusieurs classes).
- **Test système** : Tester le logiciel complet

Défi des tests unitaires

Difficile de tester de *manière unitaire* les classes dépendant d'autres classes.

Différentes manières d'une classe `A` de dépendre d'une classe `B`

- **dépendance par composition** : `A` possède un attribut de type `B`
- **dépendance par héritage** : `A` étend `B`
- **dépendance de méthode** : une méthode de `A` appelle une méthode de `B`.
- **dépendance par transitivité** : `A` dépend d'un autre objet de type `C` qui dépend d'un objet de type `B`

Il y a plusieurs manières d'injecter une dépendance entre deux classes.

Un exemple pour illustrer la problématique des dépendances

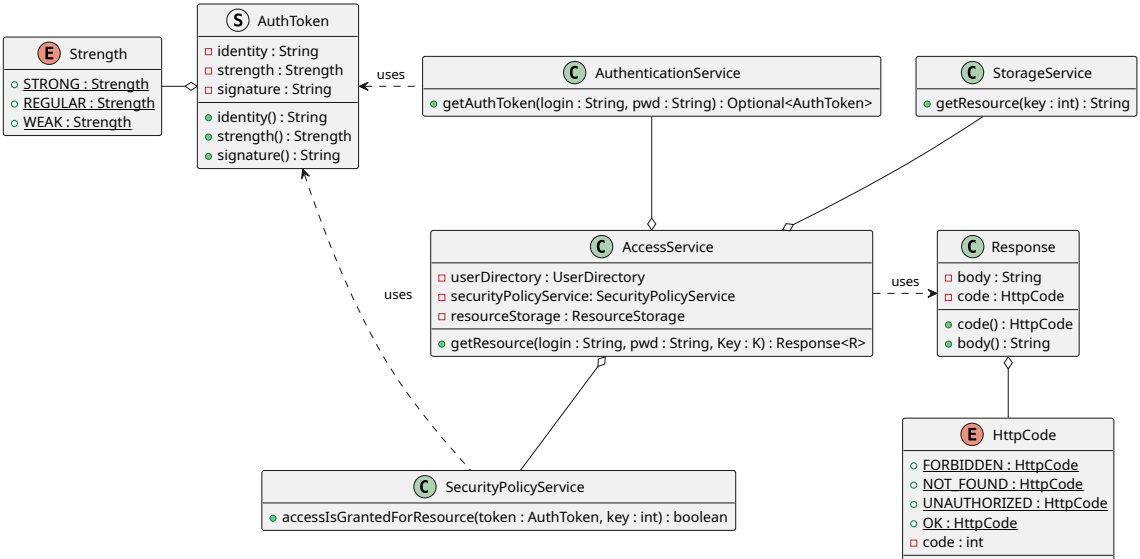
Service d'accès à des ressources contrôlé

On a un **service** d'accès à des ressources qui passe par trois services :

- un service d'**authentification** qui génère à partir d'un couple identifiant/mot de passe un jeton d'accès si le couple est correct.
- un service de **politique de sécurité** qui étant donné la clé d'une ressource et un token d'accès répond si l'accès à la ressource est donné.
- un service de **stockage** qui étant donné une clé donne le contenu de la ressource associé.

Comment tester la classe permettant l'accès qui reçoit une requête (identifiant, mot de passe, clé) donne la ressource si l'accès est autorisé ?

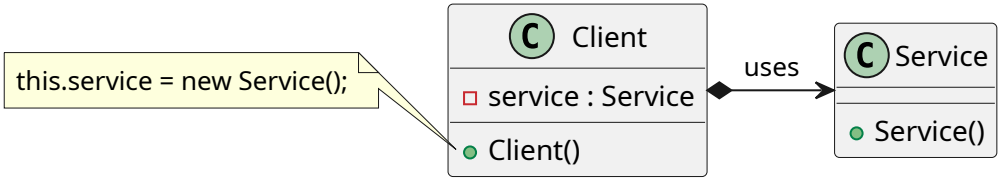
Exemple : comment tester `AccessService` ?



Section 2

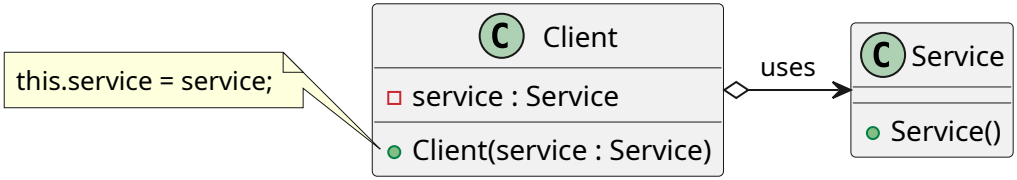
Différentes manières de gérer l'injection de dépendance

Dépendance sans injection



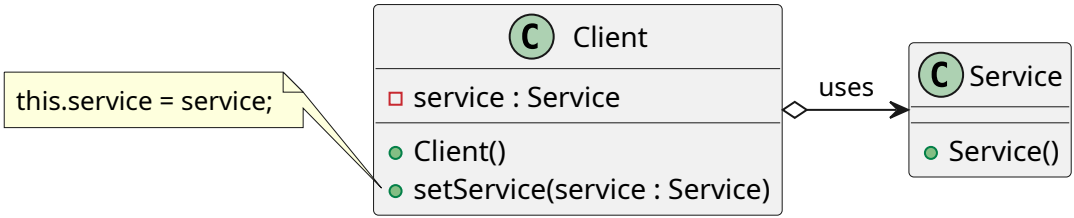
- la classe `Client` s'occupe elle-même de créer l'objet dont elle dépend dans le constructeur ;
- c'est généralement une composition, car l'instance ainsi créée dépend entièrement de l'instance qui l'a créée ;
- le désavantage est que si on souhaite changer la dépendance on doit changer le code de la classe.

Injection à l'instanciation



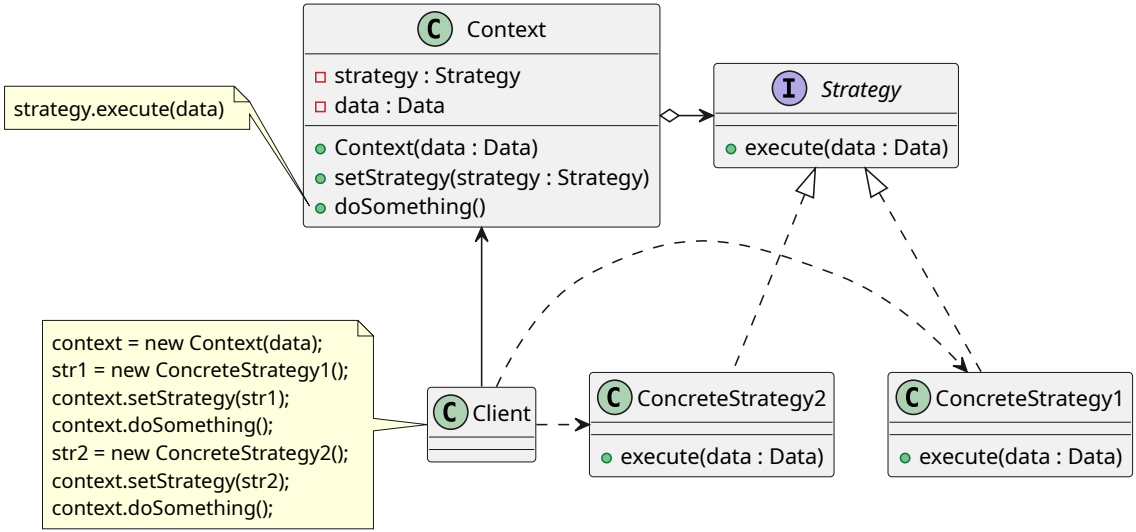
- la classe `Client` demande à la construction qu'on lui donne un objet déjà construit ;
- c'est généralement une agrégation, car on peut utiliser le même service pour plusieurs clients ;
- l'avantage est qu'on peut facilement donner une instance d'une autre classe (d'un type compatible) et donc changer la dépendance.

Injection via un Setter



- la classe `Client` permet de changer sa référence au service qu'elle utilise via un *setter* ;
- c'est généralement une agrégation, car on peut utiliser le même service pour plusieurs clients ;
- l'avantage est qu'on peut changer la dépendance pendant l'exécution (patron de conception *strategy*).

Patron de conception Stratégie



Patron de conception Stratégie

Intention

Définir une famille d'algorithmes, encapsuler chacun d'entre eux et les rendre interchangeables.

Analogie

Pour vous rendre à l'aéroport, vous pouvez prendre

- le bus,
- appeler un taxi ou
- enfourcher votre vélo.

Ce sont vos stratégies de transport et vous en choisissez une en fonction de vos besoins.

Quand utiliser le patron Stratégie ?

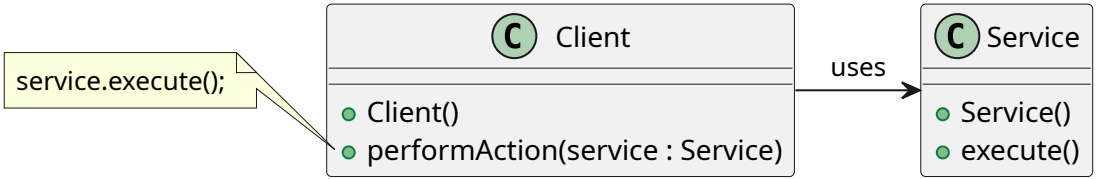
Cas d'utilisation

Une classe définit un comportement spécifique avec plusieurs manières de le réaliser.

Solution

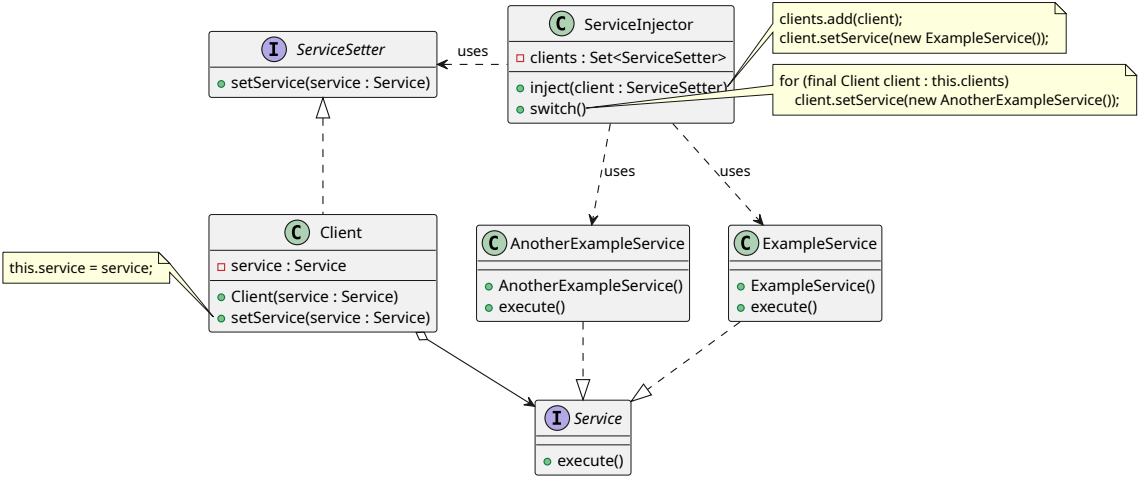
- Séparer les différentes manières de réaliser le comportement de la classe en classes séparées appelées stratégies (partageant la même interface).
- La classe originale (le contexte) garde un attribut qui garde une référence vers une des stratégies.
- Plutôt que de s'occuper de la tâche, le contexte la délègue à l'objet stratégie associé.
- Le contexte n'a pas la responsabilité de la sélection de l'algorithme adapté, c'est le client qui lui envoie la stratégie.

Injection lors d'un appel de méthode



- la classe `Client` demande lors d'un appel de méthode qu'on lui donne un objet déjà construit ;
- ce n'est ni une agrégation, ni une composition, car une instance de `Client` ne stocke pas de référence vers une instance de `Service` ;
- le désavantage est qu'on doit donner une instance de `Service` à chaque appel de méthode.

Injection via une interface



Injection via une interface

Avantages :

- découplage : l'utilisation d'interface permet à l'injecteur de ne pas connaitre la classe concrète du service ;
- possibilité de changer la dépendance de tous les clients facilement.

Désavantages :

- création de nombreuses classes qui peuvent compliquer le code.

Solution

Utiliser un framework logiciel permettant d'injecter facilement des dépendances :

- Spring Framework
- Guice

Section 3

Inversion de dépendances

Sens “naturel” des dépendances entre classes

Architecture classique :

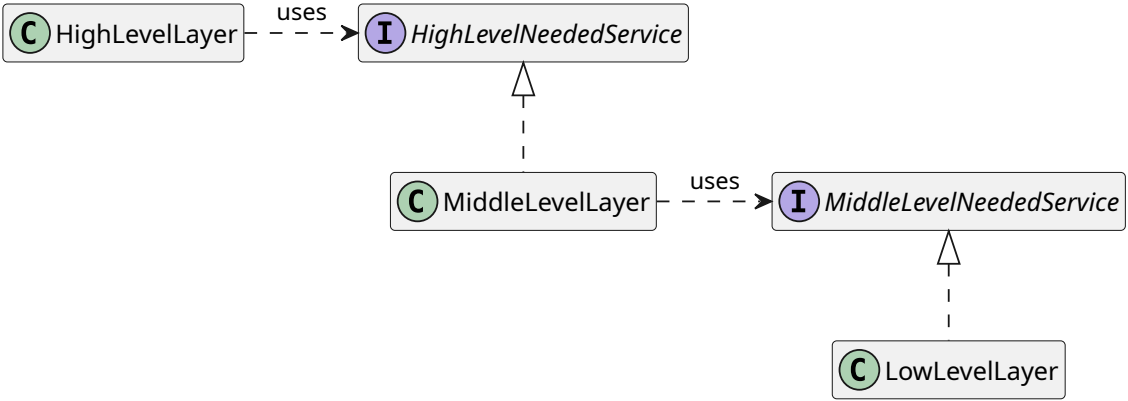
Problème

Les modules de haut niveau (couche métier) dépendent des modules de bas niveau (couche technique : stockage, graphique, ...) alors que le haut niveau change moins régulièrement que le bas niveau.

Solution

Inverser la dépendance en définissant pour chaque niveau le service qu'elle a besoin sous la forme d'une interface.

Inversion de dépendances

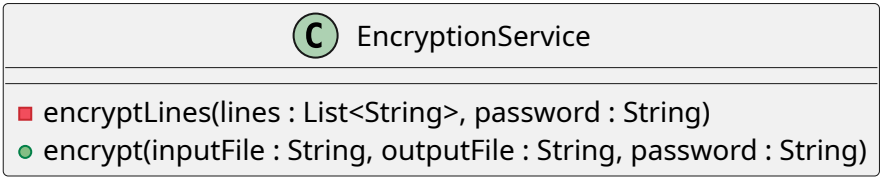


- les dépendances passent par des interfaces ;
- chaque niveau définit sa manière d'accéder au service (avec son niveau d'abstraction)

Classe EncryptionService

```
public void encrypt(String inputFile, String outFile, String pwd) {
    List<String> readLines = new ArrayList<>();
    try(FileReader fr = new FileReader(inputFile);
    BufferedReader br = new BufferedReader(fr)) {
    while (br.ready()) {
      readLines.add(br.readLine());
    }
    List<String> writtenLines = encryptLines(readLines, pw);
    try (FileWriter fw = new FileWriter(outFile);
      BufferedWriter bw = new BufferedWriter(fw)) {
      for (String line : writtenLines) {
        bw.write(line);
      }
    }
```

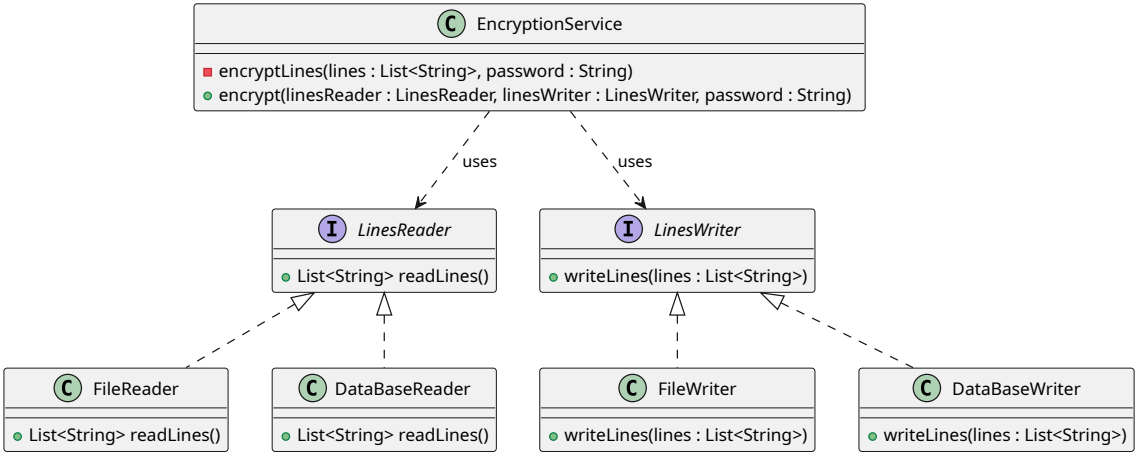
Classe `EncryptionService`



Problème

La classe `EncryptionService` dépend de considération de bas niveau : stockage dans des fichiers.
$\Rightarrow$ il est difficile de réutiliser la classe avec une autre méthode de stockage (base de données)

Solution d'inversion de dépendances `EncryptionService`



$\Rightarrow$ le niveau d'abstraction est adapté à `EncryptionService`

Inversion de dépendances

Principes :

- Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau ;
- Les abstractions ne doivent pas dépendre des détails.

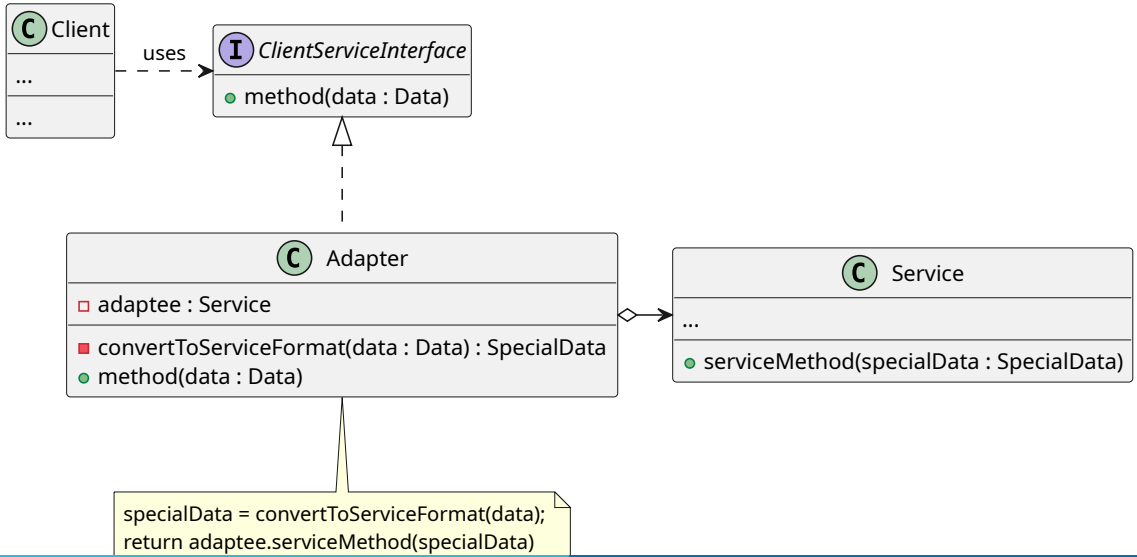
Avantages :

- Flexibilité ;
- Facilité de test : on peut utiliser des *mocks* plus facilement ;
- Réduction du couplage.

Inconvénients :

- Complexité accrue : plus de classes (par exemple patron de conception *adapter*) ;
- Sur-abstraction : Un excès d'abstractions peut rendre moins lisible les interactions entre les différentes parties du code.

Patron de conception *adapter*



Patron de conception *adapter*

Intention

Permet de faire collaborer des objets ayant des interfaces normalement incompatibles.

Utilisation du patron

Classe déjà écrite qui délègue certaines opérations à une autre classe. Changer la classe déléguée en utilisant une autre classe pas tout à fait compatible (opérations similaires, mais pas la même interface).
Solution :

1. Créer une interface pour les opérations déléguées
2. Créer un adaptateur qui convertit l'interface de votre nouvelle classe

Avantages et inconvénients

Avantages :

- Découpler l'interface ou le code de conversion des données, de la logique métier du programme (SRP).
- Permet de réutiliser du code existant sans le modifier (respect OCP)

Désavantage :

- Peut rajouter de la complexité inutile dans le code (parfois plus simple de recoder la classe service).

Section 4

Gestion automatisée des dépendances : Guice

Automatisation de la gestion des dépendances

Pourquoi automatiser la gestion des dépendances ?

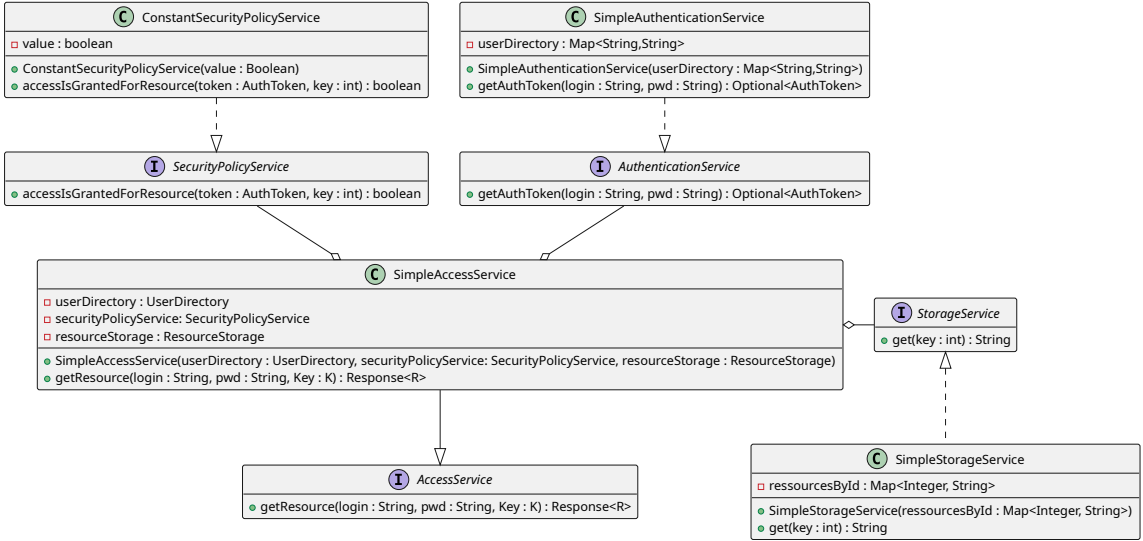
- Découpler la création de l'utilisation d'objets services
- Permettre une reconfiguration plus facile de l'application

Comment automatiser ?

avec des frameworks comme :

- Spring : *framework* d'inversion de contrôle pour la plate-forme Java (avec extension pour Jakarta EE)
- Guice : *framework* léger d'injection de dépendance

Exemple avec des interfaces



Utilisation de Guice pour l'injection via constructeur

Annoter les constructeurs pour lesquels on souhaite automatiser la dépendance avec `@Inject` :

```
public class Client{
    private InterfaceService service
    @Inject
    public Client(InterfaceService service){
        this.service = service;
    }
}
```

`InterfaceService` devra être une interface qui sera liée via Guice à une classe concrète l'implémentant.

Illustration de l'injection via constructeur

```
@Inject
public SimpleAccessService(SecurityPolicyService securityPolicy,
                           AuthenticationService authentication,
                           StorageService storage) {
        this.securityPolicyService = securityPolicy;
        this.authenticationService = authentication;
        this.storageService = storage;
}
```

Les trois classes seront liées à des classes concrètes.

Utilisation de Guice

Créer une classe de configuration qui liera des interfaces (utilisées dans les constructeurs) à des classes concrètes.

```
public class AccessModuleConfiguration extends AbstractModule {
    @Override
    protected void configure() {
        bind(InterfaceService.class).to(ConcreteClassService.class);
    }
```

Illustration des liaisons sur l'exemple

```
public class AccessModuleConfiguration extends AbstractModule {
    @Override
    protected void configure() {
        bind(SecurityPolicyService.class)
            .to(ConstantSecurityPolicyService.class);
        bind(StorageService.class)
            .to(SimpleStorageService.class);
        bind(AuthenticationService.class)
            .to(SimpleAuthenticationService.class);
        bind(AccessService.class)
            .to(SimpleAccessService.class);
    }
}
```

Création d'objets

Il est parfois utile de pouvoir créer des objets pour alimenter les constructeurs.

On annote le paramètre du constructeur pour lequel on souhaite que Guice automatise la création d'objet.

```
@Inject
Constructor(@Named("NomObjet") Object o){ /* ... */ }
```

On crée une méthode dans le Module de Guice permettant de créer cet objet

```
public class MyModule extends AbstractModule {
  @Provides
  @Named("NomObjet")
  public Object provideObject() {
    return new Object;
  }
```

Illustration de la création d'objets

```
public class SimpleAuthenticationService
    implements AuthenticationService {
    @Inject
    public SimpleAuthenticationService(
            @Named("UserMap") Map<String, String> userDirectory) {
        this.userDirectory = userDirectory;
    }
}
public class AccessModuleConfiguration extends AbstractModule {
    @Provides
    @Named("UserMap")
    public Map<String, String> provideUserMap() {
        return Map.of("alaboure", "password");
    }
```

Création d'objet avec Guice

Pour créer un objet utilisant des injections de dépendances, il faut :

1. Créer un `Injector` à partir d'une configuration (classe étendant `AbstractModule`) à l'aide de la méthode `Guice.createInjector` ;
2. Utiliser cet `Injector` pour créer une instance d'une classe via la méthode `getInstance`.

Illustration sur l'exemple :

```
Injector injector =
    Guice.createInjector(new AccessModuleConfiguration());
AccessService accessService =
    injector.getInstance(AccessService.class);
```

Section 5

Tests et mocking

Tests : problématique

Test unitaire : tester toutes les fonctionnalités d'une classe (en isolation) alors qu'elle interagit avec d'autres entités :

- Base de données (données persistantes),
- Service Web, outils réseaux,
- Classes avec un comportement non-déterministe,
- Classes de l'application en cours de développement.

Problèmes des dépendances

- Les classes services ne sont pas écrites ;
- Difficile de contrôler le comportement des services pour couvrir par les tests le code du client.

Quand simuler ?

- comportement non-contrôlable (non prédictible, non-reproductible) : évolutif dans le temps, configuration, réseau local, application web, ...
- application incomplète : modules non finalisés, méthodes incomplètes (parfois la classe sous test elle-même !),
- applications coûteuses, états du système difficiles à reproduire, effets de bords, données persistantes,
- difficultés à contrôler les états des services dont dépend la classe de façon à couvrir par des tests tout le code de la classe.

Comment simuler ?

- Méthodes **manuelles** : écriture de classes permettant de réaliser les tests (long et souvent fastidieux pour atteindre une couverture correcte)
- Méthodes **automatisées** : génération **automatique** de classes permettant de réaliser les tests.

Outils

- mockito : création d'objets fictifs
- wiremock : création de faux serveurs d'API HTTP

Que simuler ?

- Comportements calculatoires :
  - bons paramètres,
  - bons résultats.
- Appels des méthodes :
  - bonne méthode,
  - nombre d'appels.
- Déclenchement d'exceptions

Terminologie de la simulation pour les tests

Idée simple : simuler une classe utilisée de manière plus ou moins précise.

- *dummy* : objet sans fonctionnalité
- *stub* (bouchon) : renvoie une valeur fixée par méthode
- *fake* (simulateur) : implémentation partielle qui renvoie toujours les mêmes valeurs selon les paramètres.
- *spy* (espion) : vérifie l'utilisation des méthodes.
- *mock* (doublure) : *spy* + *fake*.

Automatisation : Mockito

Principes de Mockito :

- créer les objets *mocks* facilement (`@Mock`).
- permet aussi d'espionner des véritables instances (`@Spy`)
- décrire leur comportement (*stub de méthode*)
- à l'exécution, les interactions sont enregistrées
- à la fin des tests on interroge les objets *mock* pour savoir comment ils ont été utilisés.

Utilisation : ajout de jar dans les propriétés du projet ou via une dépendance *gradle*

Automatisation : écriture de tests (1/2)

Dans la classe de test précédée de `@ExtendWith(MockitoExtension.class)`

- Définir les objets simulés :

  ```
  mockObject = mock(Classe.class)
  ```

  ou

  ```
  @Mock
  Classe mockObject;
  ```

- Spécifier le retour lors d'un appel (*stubbing*):

  ```
  when(classe.methode(arg1,...,argn)).thenReturn(resultat);
  ```

Automatisation : écriture de tests (2/2)

- Spécifier le déclenchement d'exception :

```
doThrow(new MonException()).when(mock).methode()
```

- Vérification : nombre d'appels `times()`, pas d'appel `never()`, au moins un `atLeastOne()`, ...

```
verify(mock, times(n)).foo(arg1,...,argn)
```

- Valeurs anonymes : `anyDouble()`, `anyString()`, ... pour désigner une valeur quelconque du type.
- Spécification de plusieurs appels :

```
when(appel).thenReturn(resultat1, resultat2)
```