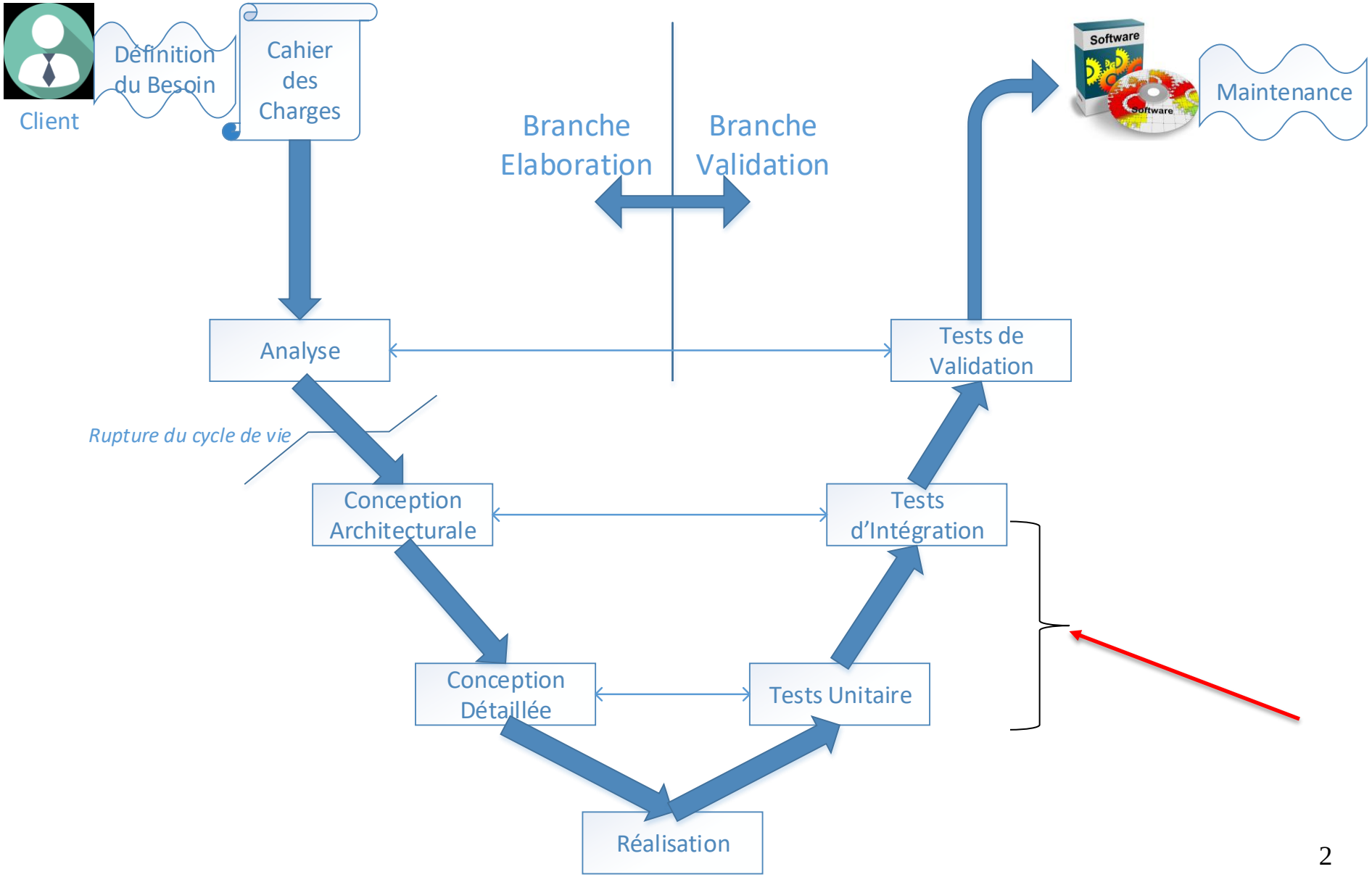
Ingénierie du Logiciel
Master 1 Informatique – 4I502

Cours 7 :
Tests d'Intégration
Tests Unitaires

Yann Thierry-Mieg
Yann.Thierry-Mieg@lip6.fr

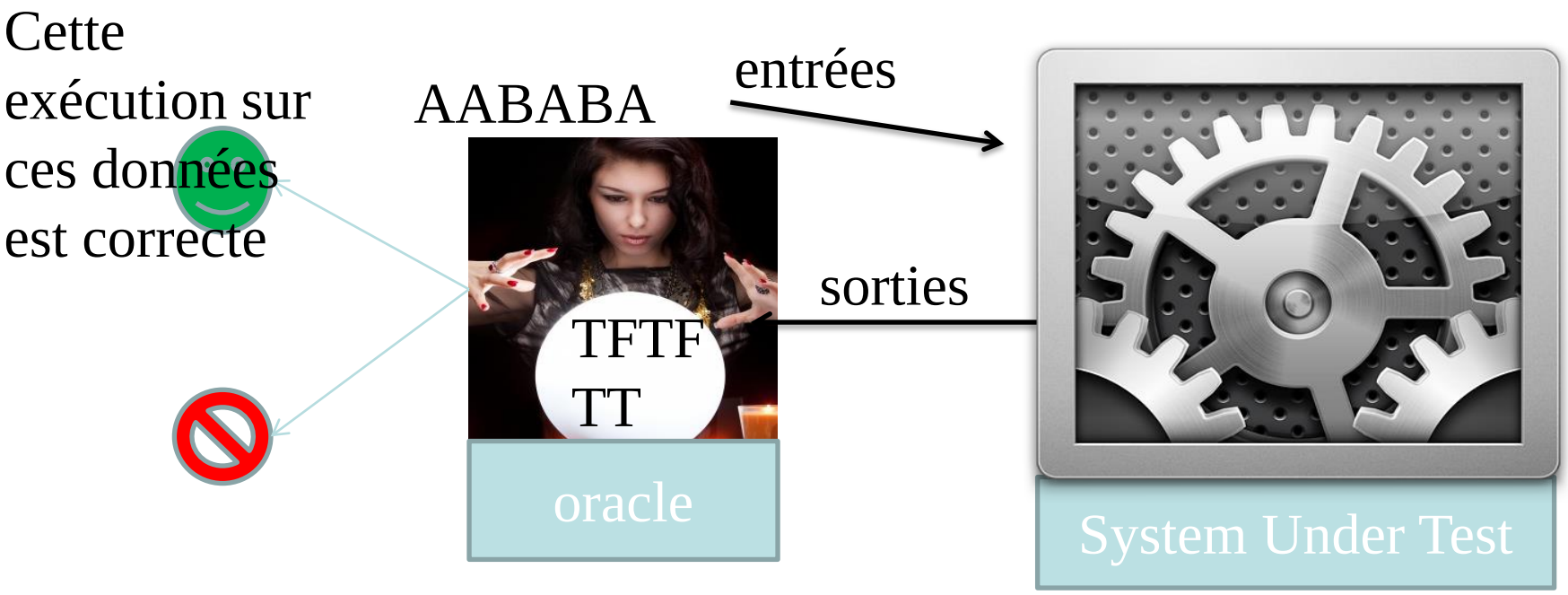
Rappel


Client
Définition du Besoin
Cahier des Charges
Branche Elaboration
Branche Validation
Software
Maintenance
Analyse
Tests de Validation
Rupture du cycle de vie
Conception Architecturale
Tests d'Intégration
Conception Détaillée
Tests Unitaire
Réalisation

Tests d’Intégration

Rappel : Validation par les tests

Principale approche industrielle pour la validation : Tester

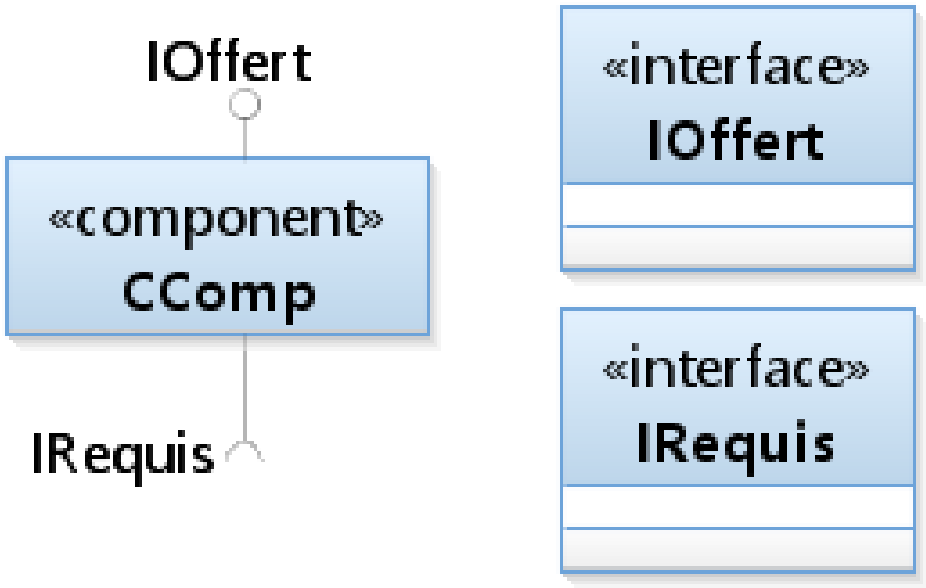


++ Confiance dans le système

-- Pas de **garantie** que le système est correct

Granularité Tests d'Intégration

- Système à tester :
  - Un composant en isolation
- Entrée :
  - Séquence d'invocations aux opérations offertes
- Sortie :
  - Réponses aux invocations + invocations du composant sur ses dépendances

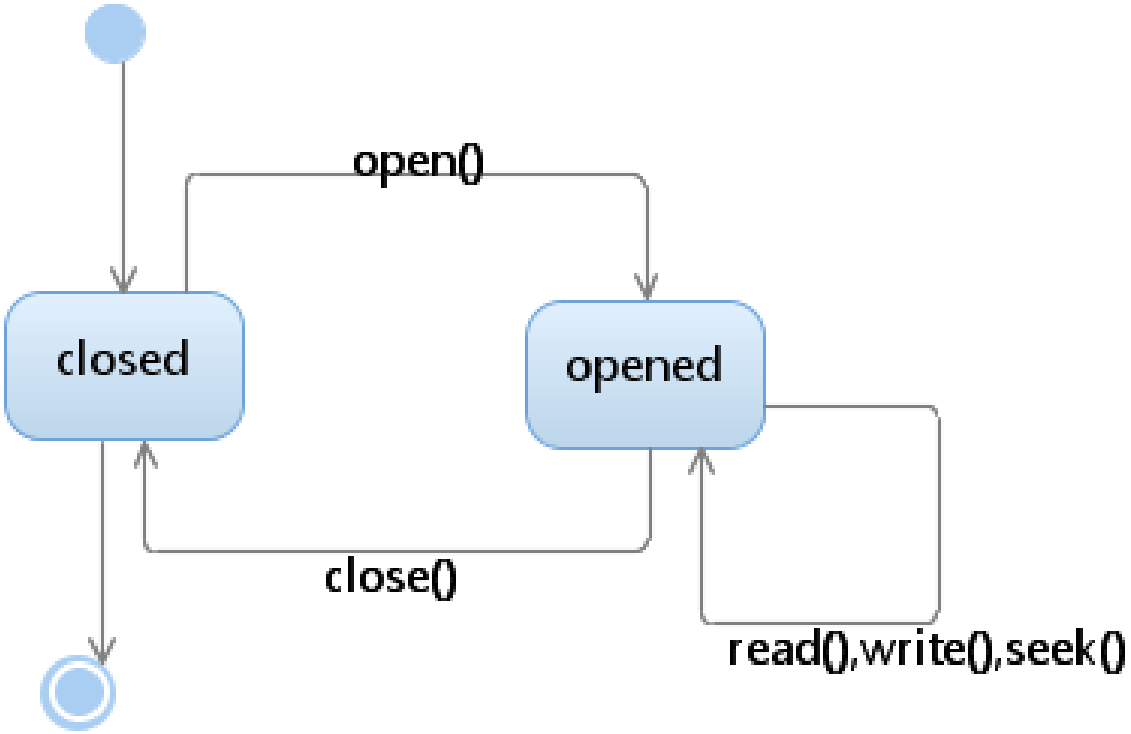


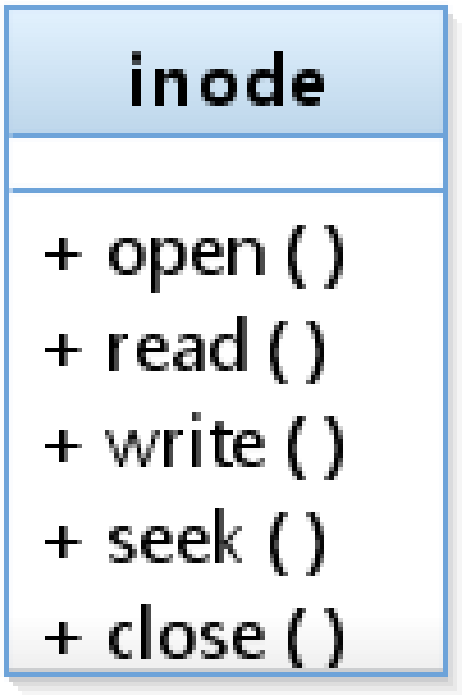
Objectifs des T.I.

- Figer les interfaces
  - Notion de contrat offert par chaque composant
  - Permettre le développement en parallèle des composants
- Assurer que l'assemblage final se passera bien
  - Spécifier et capturer les échanges entre composants
  - Préparer l'intégration des composants
- Test d'intégration
  - Exécutable != Test validation
  - Artefact technique du développement
- Test :
  - Isoler le composant (maîtrise de ses dépendances)
  - Capturer le *contrat* d'utilisation du composant

Un contrat : Protocol State Machine

- Les composants sont en général munis d'un état interne
  - Séquence d'invocations nécessaire pour tester son comportement
  - Contrat, capturable par un *protocol state machine* ou d'autres mécanismes
  - Cf aussi chorégraphie de webservices





Capturer le comportement via les tests

- Capturer via le test
  - Quelles métriques de couverture ?

Métriques et Qualité

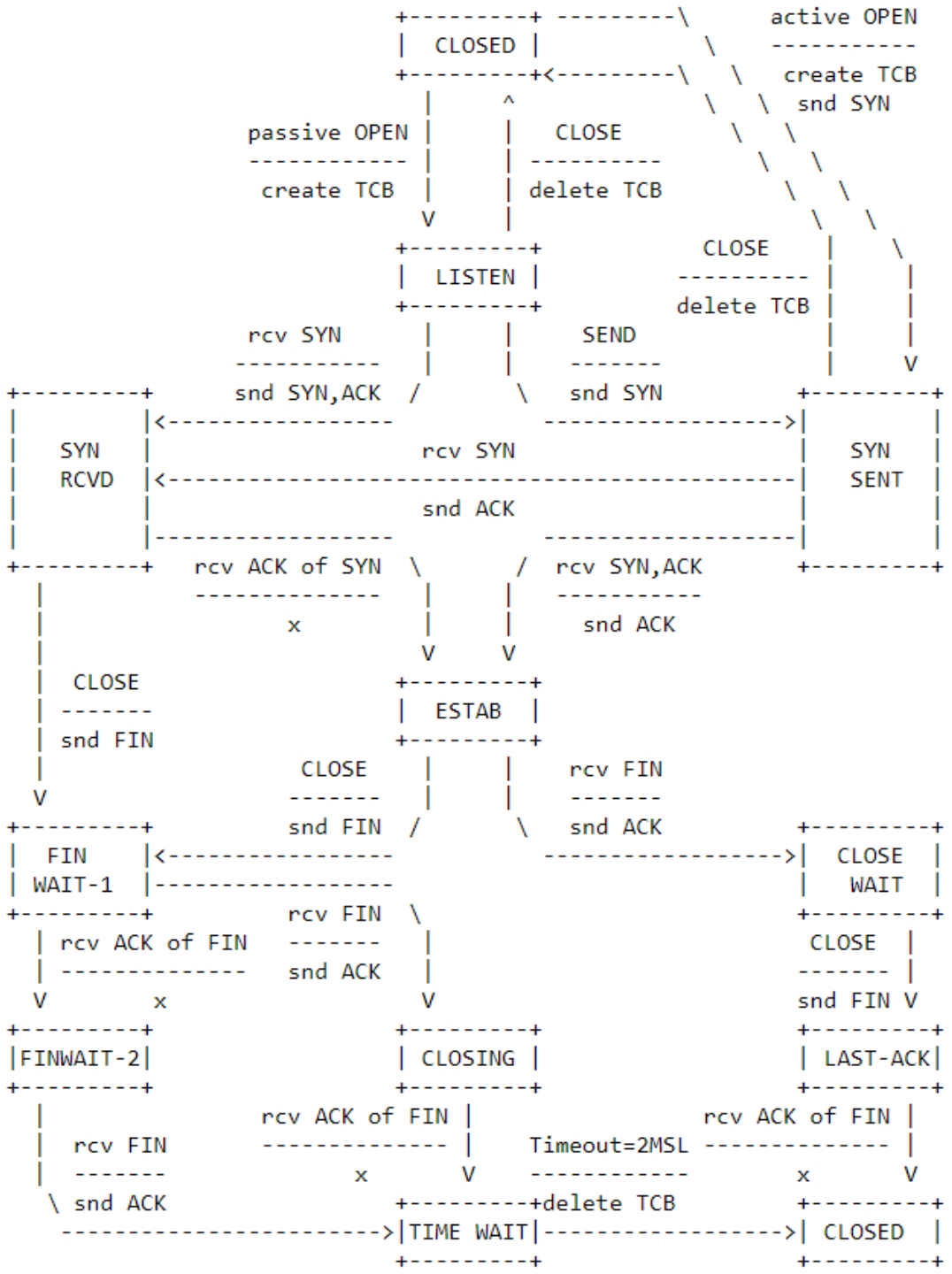
- La qualité est assurée par un Plan de Qualité Logiciel
  - Contrôle -> Analyse -> Amélioration du PQL -> reboucle
- PQL :
  - Process/Méthode à appliquer, parfois très strictement
  - Introduction de métriques de suivi de l'évolution logicielle
    - Indicateurs objectifs, dont l'évolution ou la distribution peut révéler des soucis
  - Evaluation des risques
    - Evaluation du risque : mineur -> critique
    - Lié aux exigences du projet
    - Mise en place de mesures de mitigation du risque
- Les process « lourds »
  - Certifications pour marchés publics, secteur critique
  - CMMI, ISO 9002, 9015, 15504,…

Capturer le comportement via les tests

- Capturer via le test
  - Quelles métriques de couverture ?
    - Couverture des états de contrôle
    - Couverture des transitions possibles **et impossibles**
    - Utilisation de différents jeux de données
      - /!\ le test reste par nature incomplet
    - Certains standards définissent des machines à état, e.g. IEEE TCP

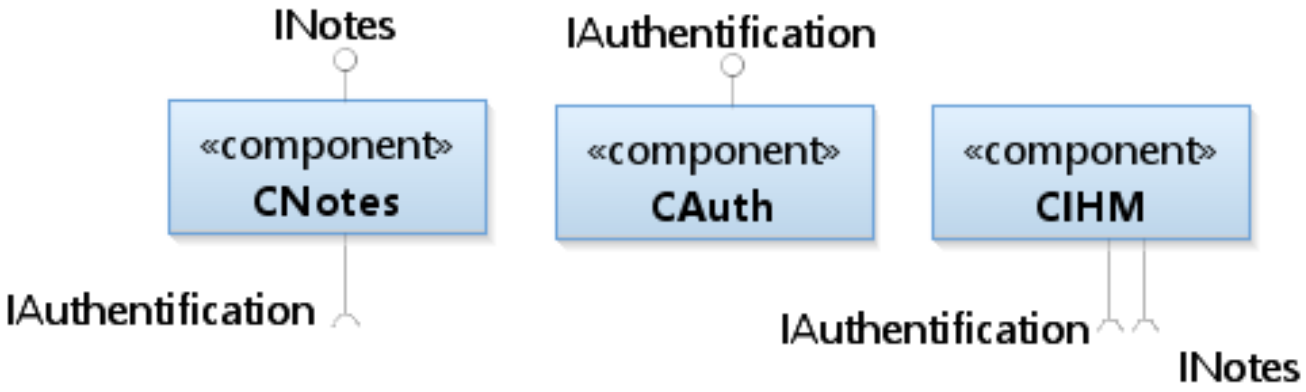
Exemple TCP

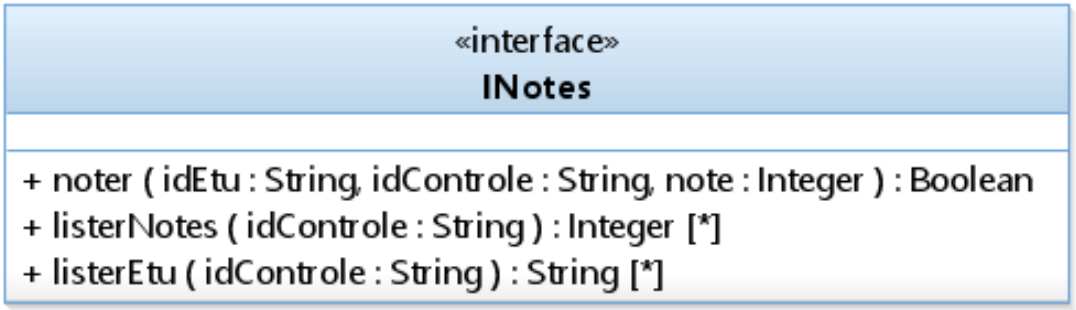
RFC 973, 1981 : Transmission Control Protocol

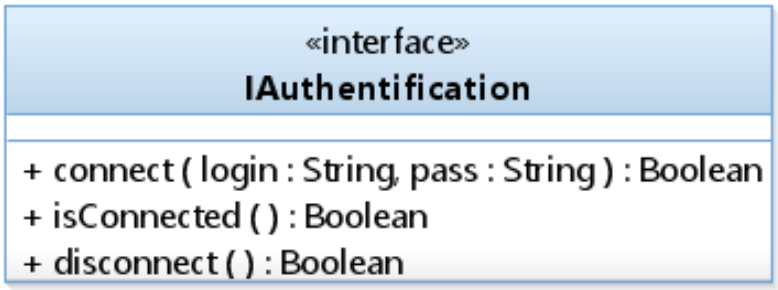


Des Séquences Inter-Composant aux Tests

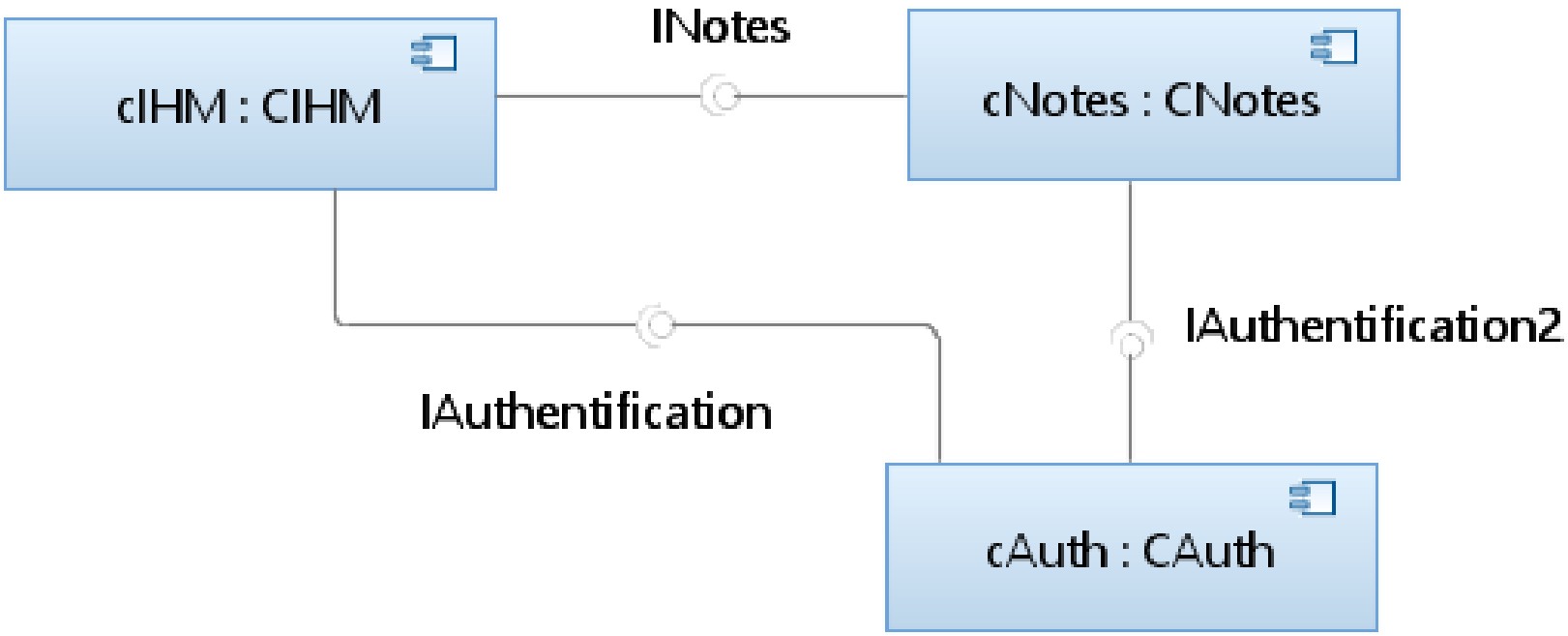
- On a construit en Conception Architecturale :
  - Composants, interfaces, interactions



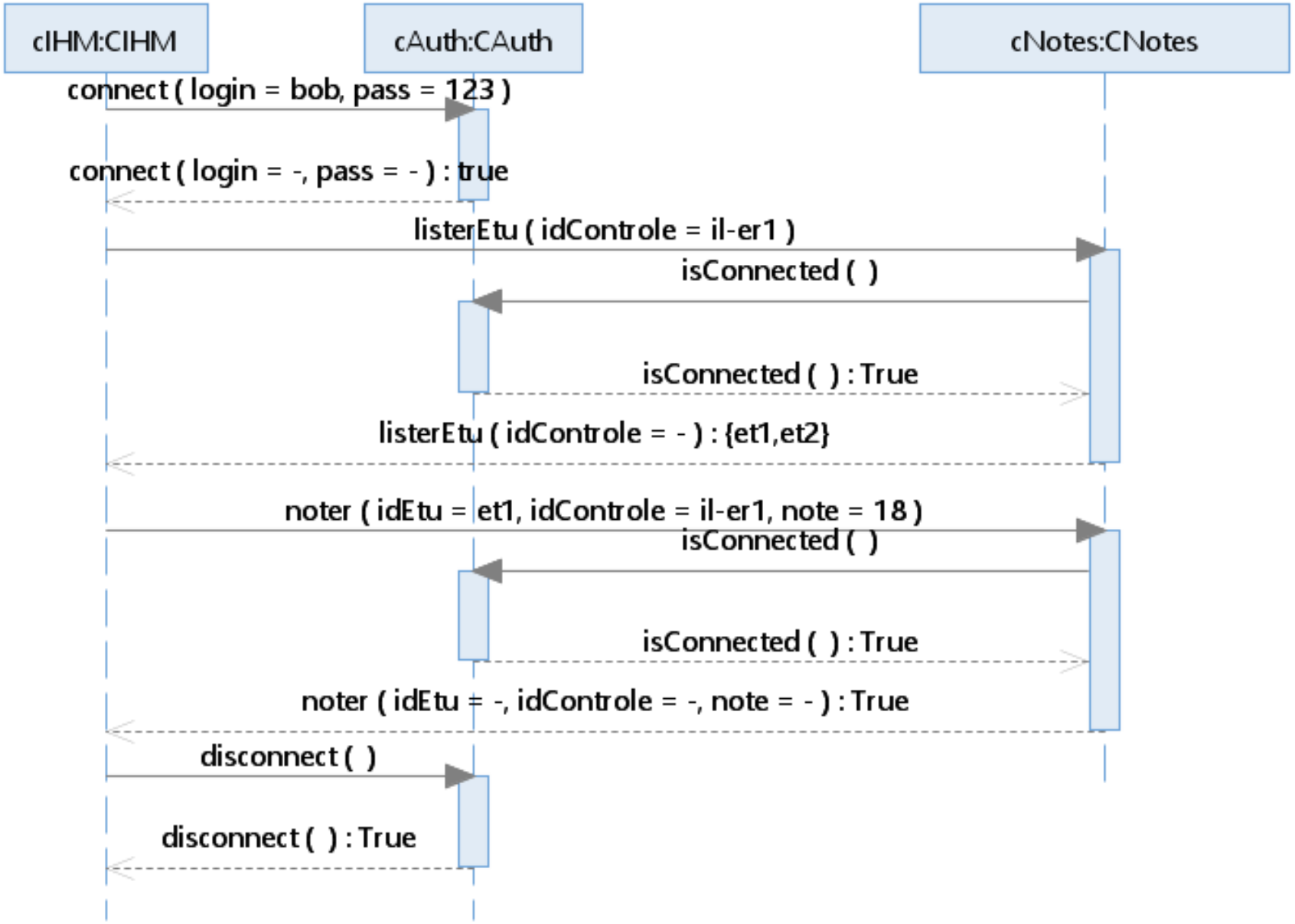




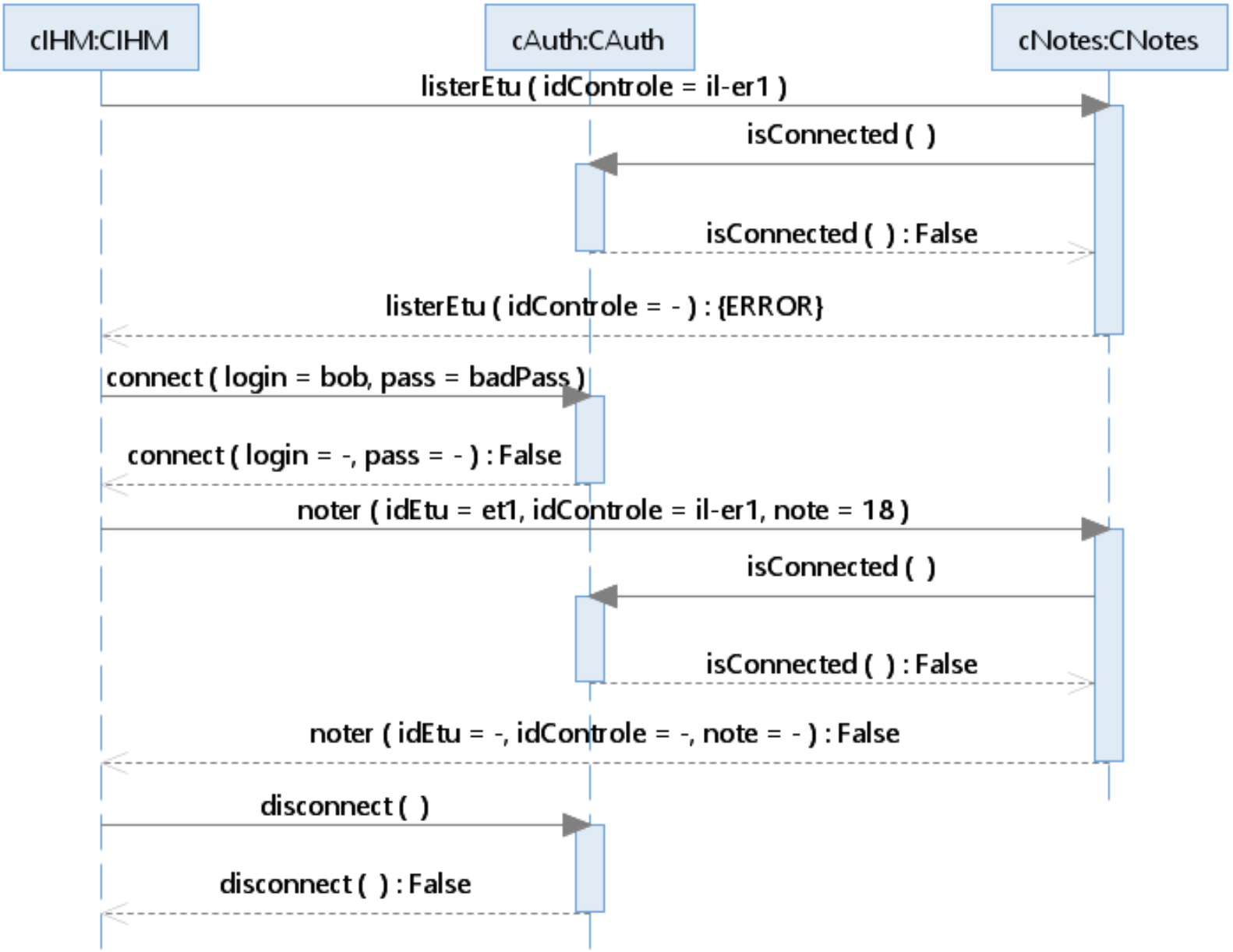
Instanciation nominale



S1 : Séquence OK



S2 : Séquence Problème

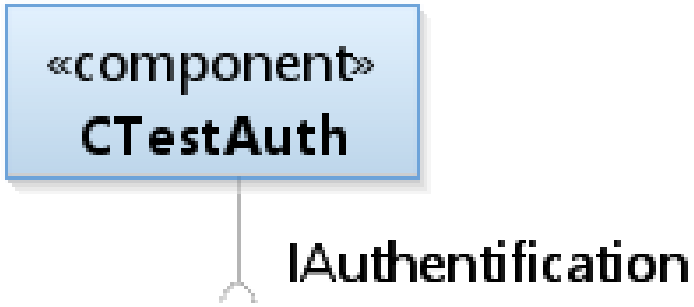


Séquences de test ?

- On projette l'interaction sur une ligne de vie
  - Par exemple : Cauth
- S1 :
  - connect(bob,123) -> True
  - isConnected() -> True
  - isConnected() -> True
  - disconnect() -> True
- S2 :
  - isConnected() -> False
  - connect(bob,badPass) -> False
  - isConnected() -> False
  - disconnect() -> False
- Ce sont directement des tests d'integration !
  - Implantable en e.g. JUnit avec la Factory de composant.

Configuration de test

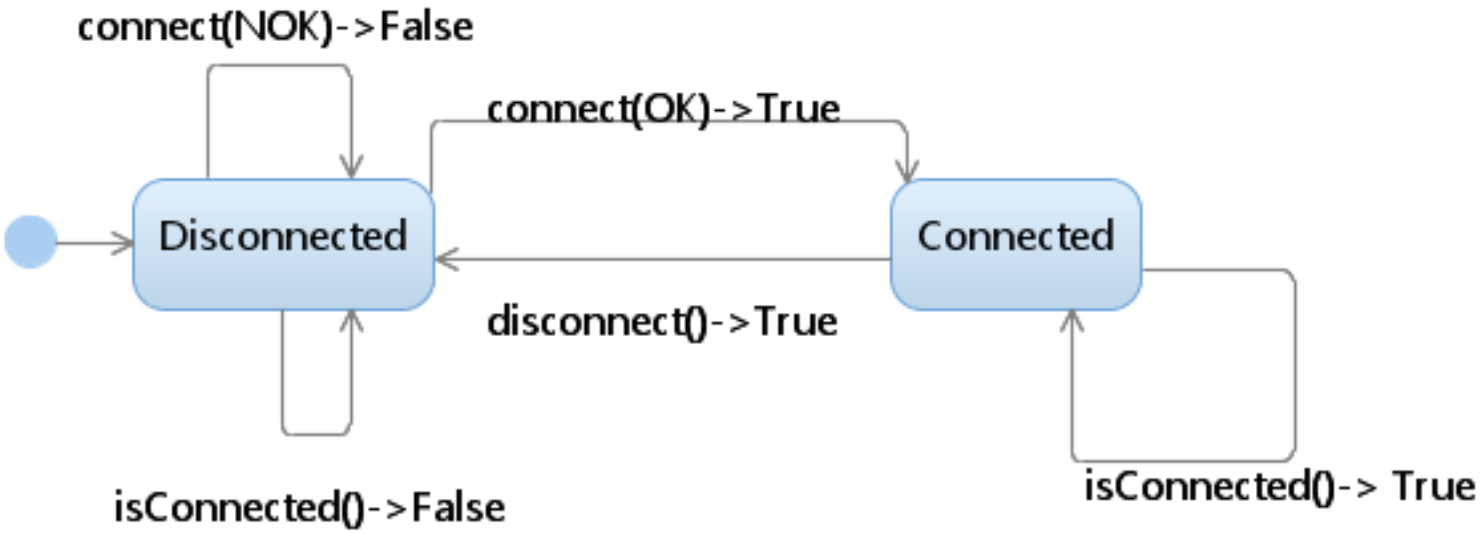
- On introduit un composant : CTestAuth
  - Il va héberger les tests Junit



  - On prévoit une configuration de test :

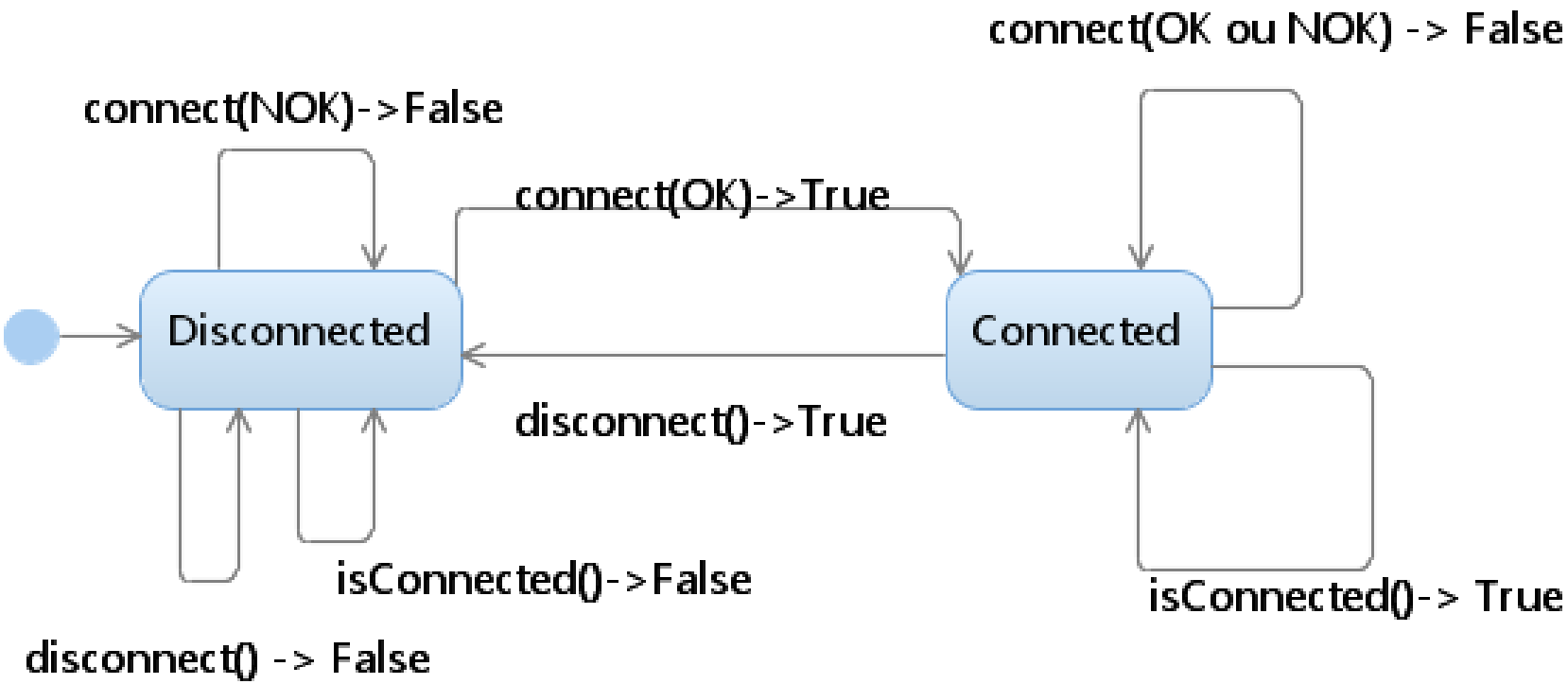
Etats de l'authentification ?

- Raisonner sur la machine à états => être plus complet
  - Connect dans l'état « connected » ?
  - Disconnect dans l'état « disconnected » ?



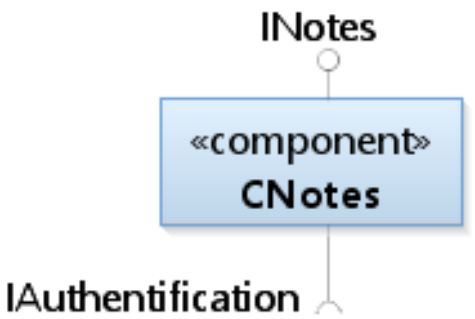
Etats de l'authentification ?

- Raisonner sur la machine à états => être plus complet
  - Connect dans l'état « connected » ?
  - Disconnect dans l'état « disconnected » ?
- Couvrir par les tests toutes les transitions si possible



Gestion des dépendances

- On considère CNotes



- Séquences projetées :
- S1 :
  - listerEtu(il-er1) -> {et1,et2}
    - + invocation isConnected()->True
  - setNote(et1, il-er1,18) -> True
    - + invocation isConnected()->True
- S2 :
  - listerEtu(il-er1) -> {ERROR}
    - + invocation isConnected()->False
  - noter(et1,il-er1,18) -> False
    - + invocation isConnected()->False

Problème : tester en isolation

- Le composant d'authentification (dépendance) n'est pas encore réalisé
  - De plus si on l'intégrait au test, on en serait plus en train de tester en isolation :
    - Le test échoue, à qui la faute ?

✓ e.g. Un système de paiement, un système de diffusion de messages appuyé par twitter, …

- On ne veut pas utiliser la vraie dépendance pendant les tests

- Comment permettre un test réaliste ?
  - En isolation, le système à tester = 1 composant
  - Sans instrumenter le code

Composant Bouchon

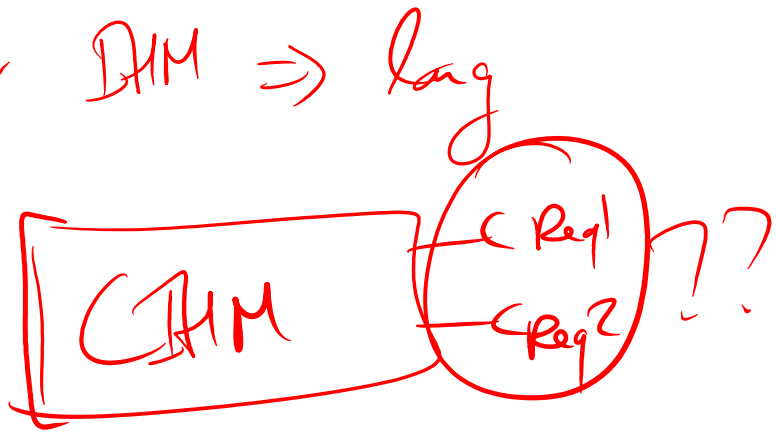
- Notion de « Bouchon » ou « Mock »
  - Simule approximativement le comportement de la vraie dépendance
  - Offre l'interface requise !
  - Doit rester simple/basique, pas de fautes dedans
    - Contient le plus souvent des réponses « en dur »
    - Une seule classe de réalisation, pas de dépendances !
  - ✓ Sert les besoins de un ou plusieurs tests
    - Mais ne permet pas d'exécuter des interactions arbitraires

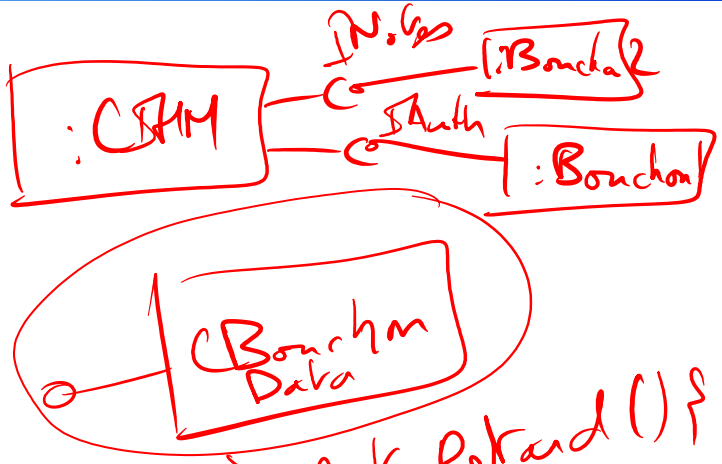
Des bouchons pour l'authentification ?

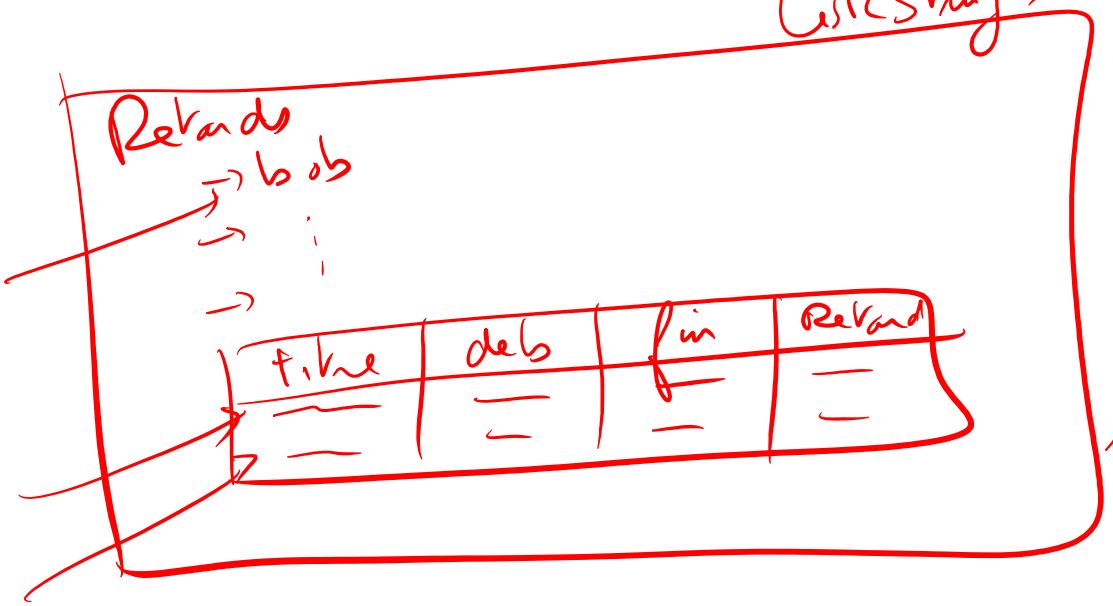
- Versions très simple :
  - CAuthTrue : connect(*)-> True, isConnected()-> True, disconnect()->True,
  - CAuthFalse : False
- Versions un peu plus riches :
  - Un booléen pour l'état, une seule paire login/password reconnue
- Ne pas faire trop complexe
  - On veut juste pouvoir utiliser CNotes sans avoir encore fait le LDAP…

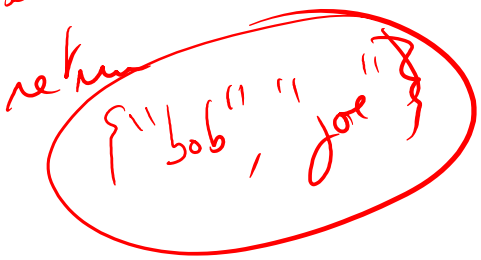
Bouchons pour IHM

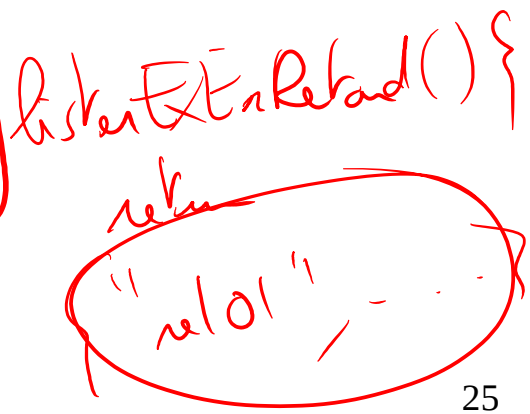
dév IHM ⇒ bug



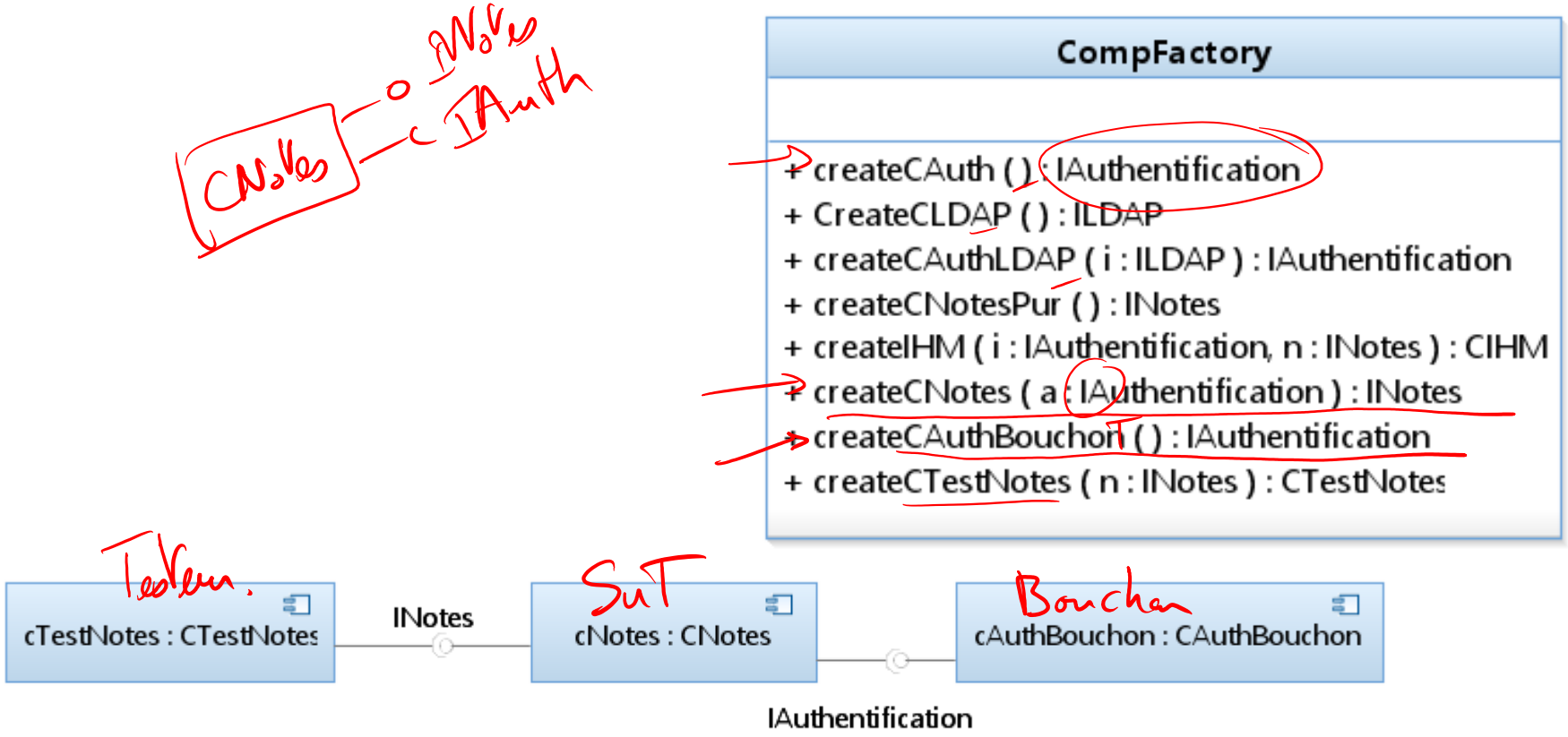








Appui sur la factory

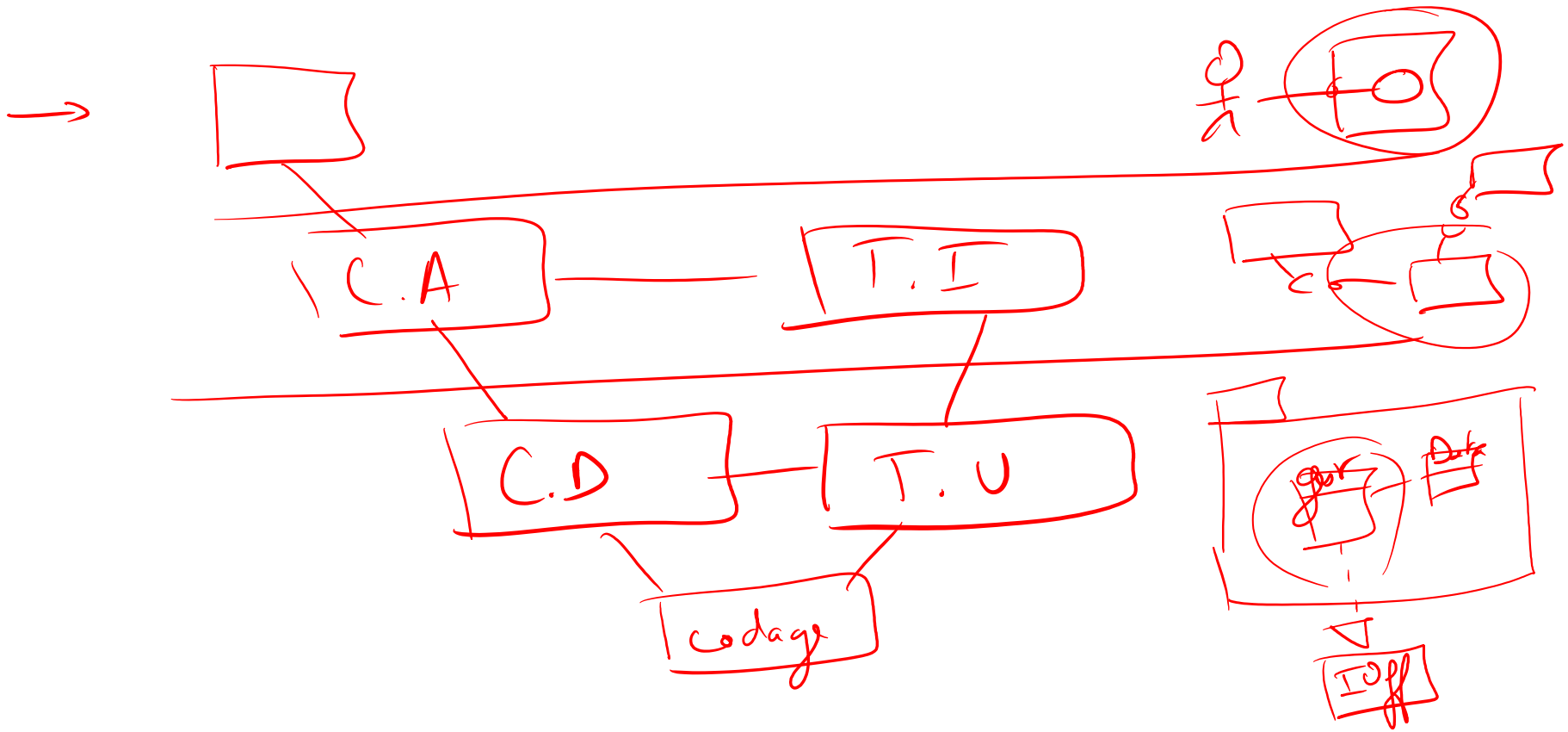



```
IAuth  a = CompFactory.createCAuthBouchon();
INotes n = CompFactory.createCNotes(a);
CTestNotes ctn = CompFactory.createTestAuth(n);
```

Tests Composant

- Frameworks existants
  - nUnit
  - TTCN 3

Tests Unitaires



→ Granularité :

→ SUT = 1 classe

T. U.

→ Métrique satisfaction ?

→ line coverage

→ exécute les tests avec instrumentation (sous debugger) ⇒ identifier les lignes de code "exercées" par les tests.

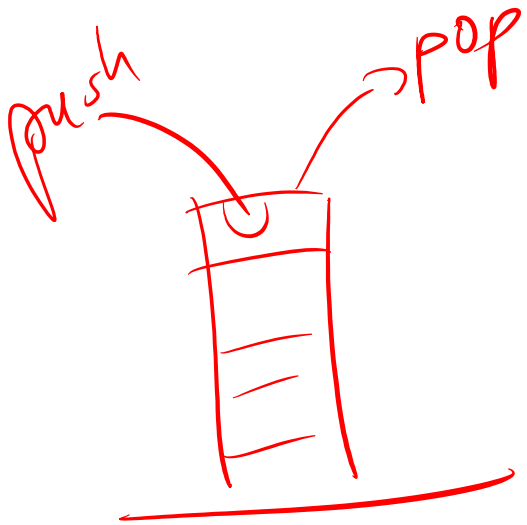
→ Chaque méthode, chaque branche if/then

→ Outils de référence existent.

→ xCoV

→ ECL - emma (eclipse + JUnit)

T.U.



Approche

→ Assume / Garantie

pré

post

→ Programmation par contrat.

pop

→ pré-condition → {non vide

→ post-condition → {taille = taille - 1

...

Conception Architecturale vs Détaillée

C.A : Inter-Composants

- Les composants sont des boites noires
  - Responsabilités de données + traitements
    - Issus des classes métier découpées + opérations du système
  - Connectés les uns aux autres via des interfaces requises/offertes
  - Séquences => un lifeline par instance de composant

C.D : Intra-composant

- Réalisation précise d'un composant
  - Comment stocker les données et implanter les traitements
  - Acteurs = les autres composants du système
  - Niveau classes, préparer la réalisation

Relation de *raffinement* entre ces descriptions.

Cohérence !

Conception Détaillée : Objectifs

- Préparer la réalisation
  - Implémentabilité sur une plateforme fixée
- Choix de conception fins
  - Langage ou technologie support
  - Structure de données, index/map, …
  - Design Patterns, schéma de BD,…

Plateformes à Composants

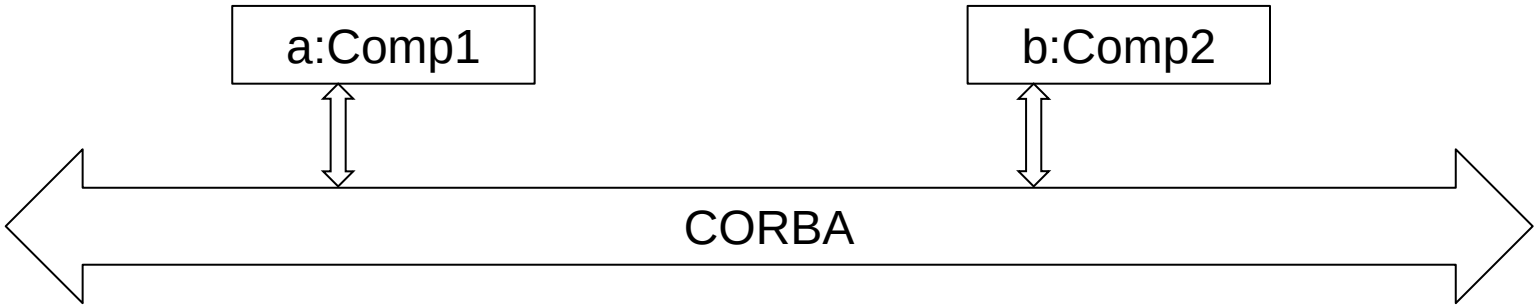
CORBA (’92)

Common Object Request Broker Architecture (Standard OMG)

Notion de Bus Logiciel

Gestion de défis technologiques de façon générique

- Langages hétérogènes
- Réseau
- Architectures matérielles hétérogènes



a souhaite invoquer : bool foo (String s) sur b

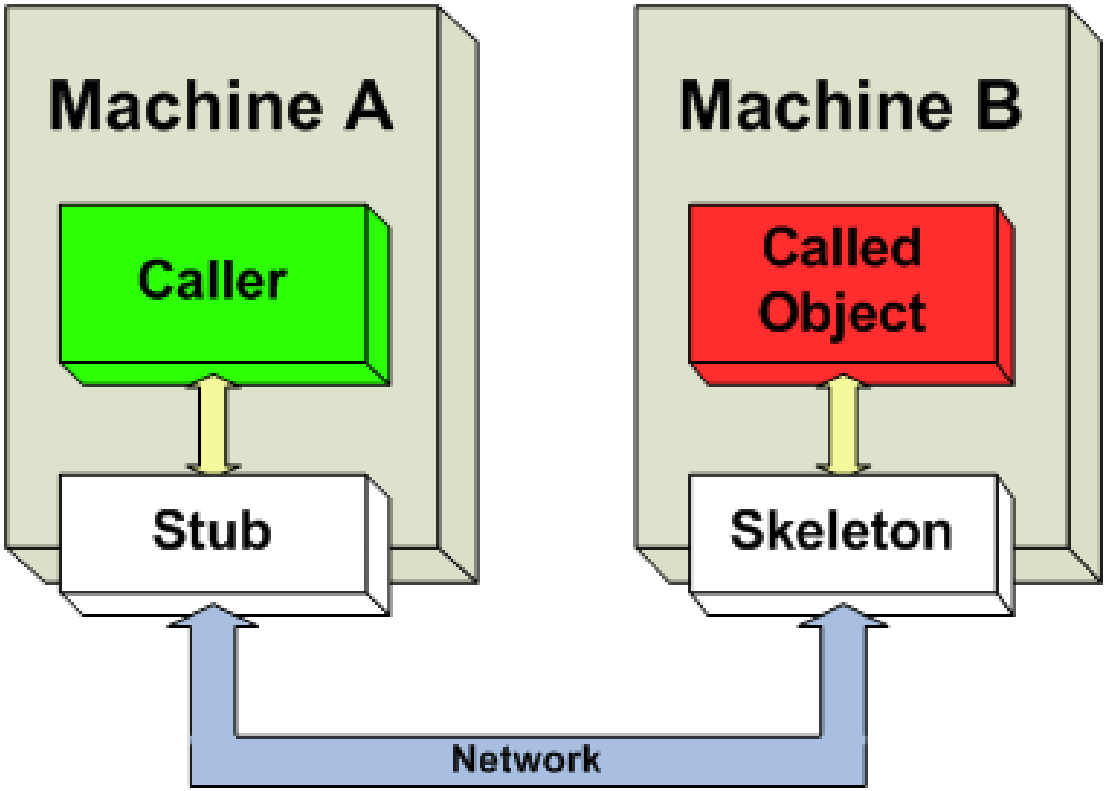
CORBA

- Une API de communication indépendante du langage
  - Déclaration des services dans un langage neutre :
    - IDL (Interface Definition Language)
  - ✓ Module (+- un namespace) + Interface(s) portant de opérations
    - Signatures uniquement
    - Cf Connecteurs Medvidovic ADL

```
module HelloApp
{
  interface Hello
  {
  string sayHello();
  oneway void shutdown();
  };
};
```

Compilation de l'IDL

- Construire pour une cible particulière
  - Stub : un *proxy* pour le client, représente localement l'objet distant
  - Skeleton : un élément côté serveur qui permet de recevoir les requêtes
  - Le même IDL peut générer du Java, du C, de l'ADA, du Fortran …



DP Proxy

Proxy : objet qui fait semblant d'être un autre objet

Par exemple le proxy réseau :de votre browser se comporte comme un gateway, internet (e.g. comme une box) mais rajoute des traitements (filtres, cache…)

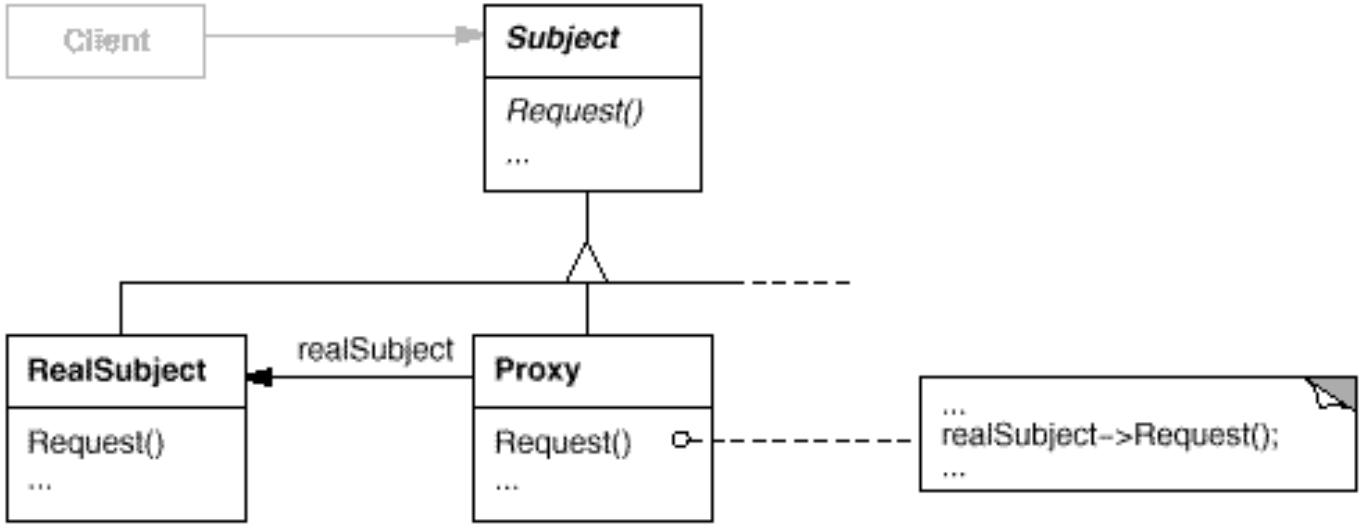
Pour le DP, le proxy est une classe qui implémente les mêmes opérations que l'objet qu'elle protège/contrôle.

Plusieurs variantes de Proxy, selon la finalité :

- ✓ Proxy Virtuel : retarder les allocations/calculs couteux
- ✓ Proxy de Securité : filtre/contrôle les accès à un objet
- ✓ Proxy Distant : objet local qui se comporte comme l'objet distant et cache le réseau
- ✓ Smart Reference : proxy qui compte les références (C, C++)

DP Proxy : Structure

GOF, GHVJ'95



Subject : interface manipulée par le client

RealSubject : un objet lourd à instancier

Proxy : retarde la création du sujet réel

Délégation particulière, où délégat (Proxy) et délégué (RealSubject) réalisent la même interface

DP Proxy Distant : Principes

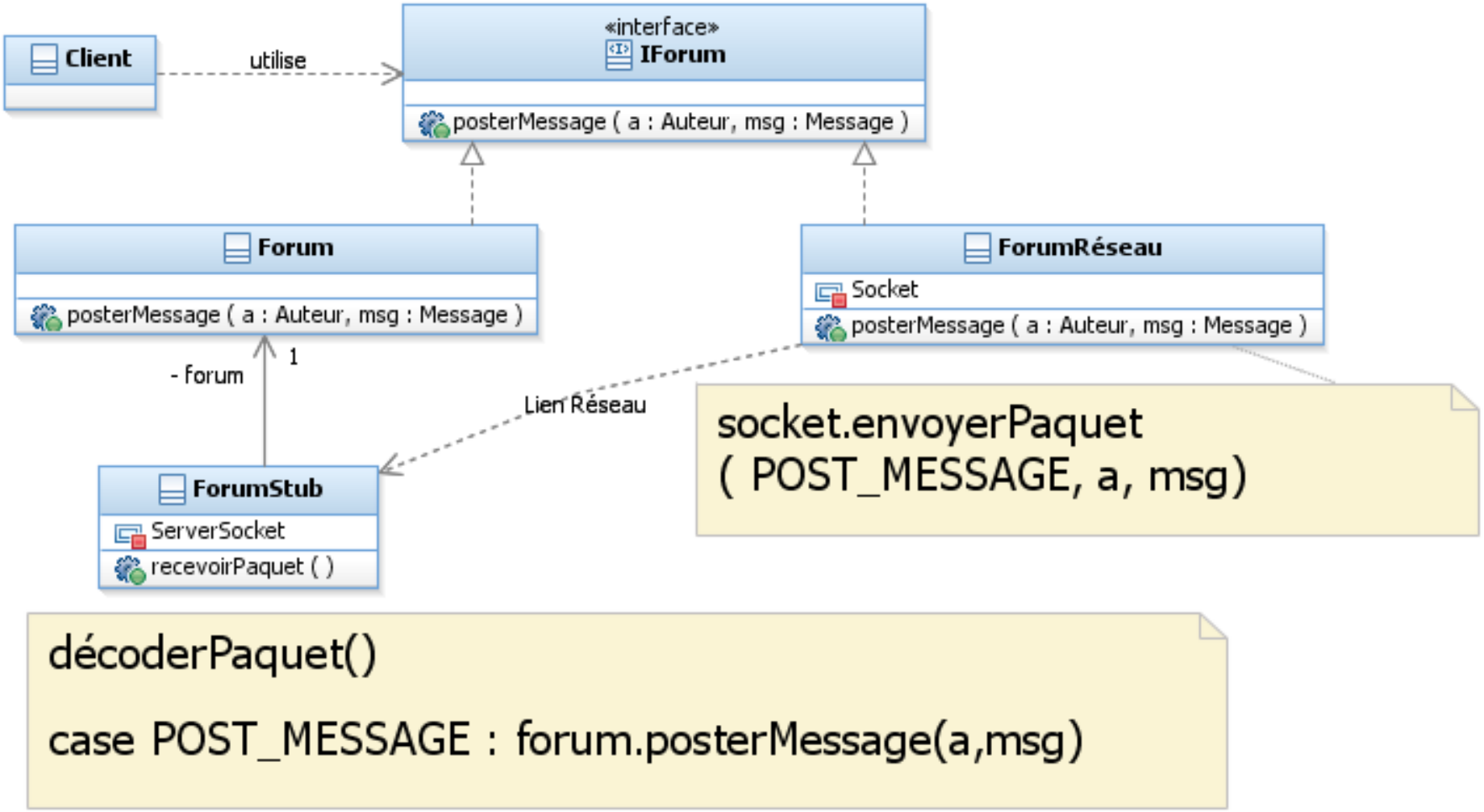
On a une application répartie sur plusieurs machines

On voudrait développer l'application sans trop se soucier de où sont physiquement stockés les objets

Proxy réseau : objet local à la machine, qui se comporte comme l'objet distant, mais répercute ses opérations sur l'objet distant via le réseau

- ✓ Comportement par délégation, mais avec le réseau interposé

DP Proxy distant



Proxy distant: conclusions

Généralise la notion de RPC (remote procedure call)

Rends transparent la localisation de objets

La réalisation du Proxy réseau et du stub suit une ligne standard

De nombreux frameworks offrent de générer cette glu automatiquement (et/ou de la cacher)

- ✓ Java RMI : remote method invocation
- ✓ CORBA…

CORBA : Services

Services offert par le bus logiciel

- Gestion des communications
  - Passage par copie des arguments, marshall/unmarshall = sérialisation
  - Gestion du réseau
  - Service de nommage ou adressage
    - Obtenir des références à des objets distants
  - ✓ Création, Destruction, configuration des composants
    - Cycle de vie des composants
  - ✓ Autres services
    - Transactions, Sécurité, Singleton, …

CORBA est un *middleware*

- fournit une abstraction des plateformes techniques

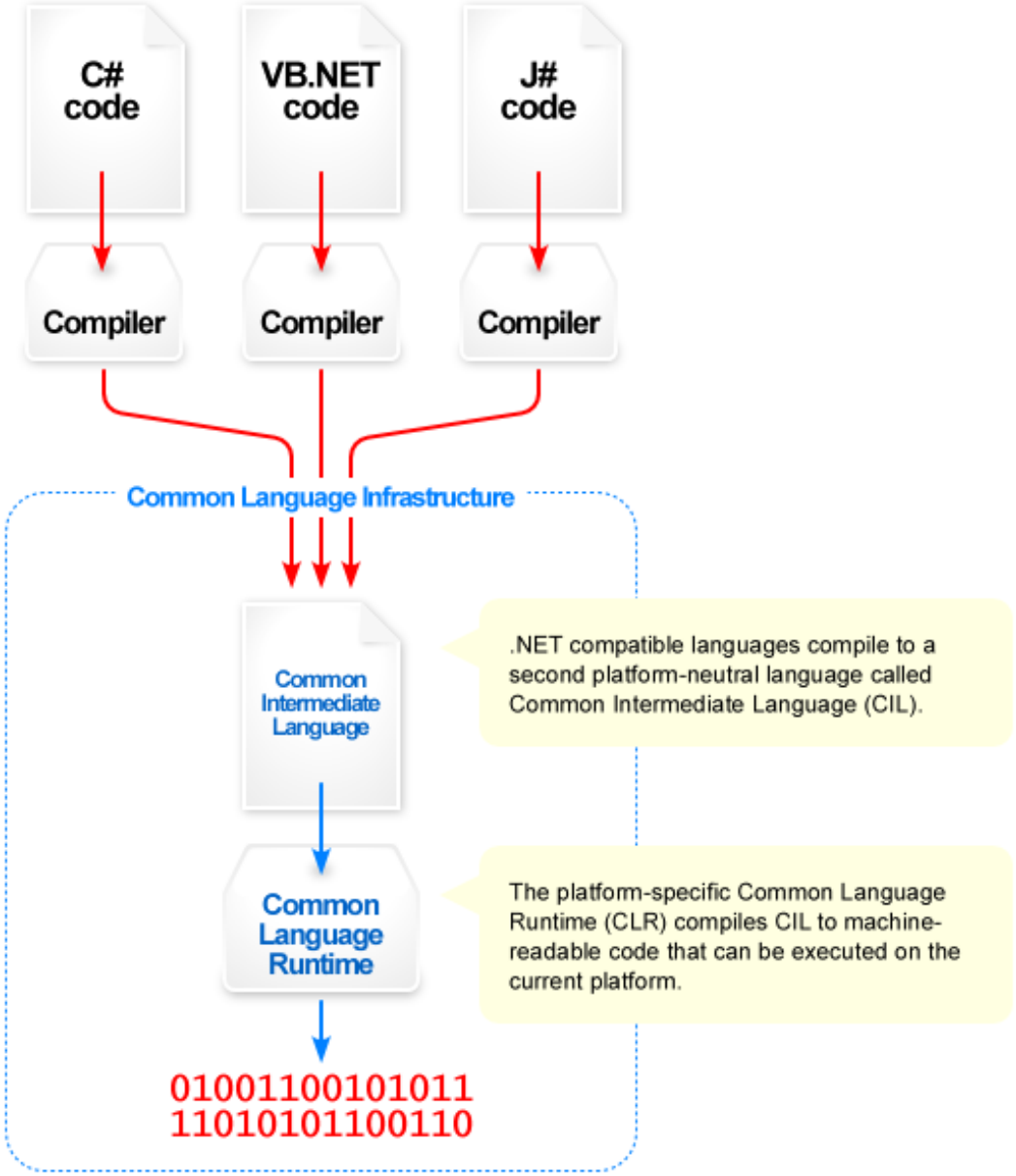
Plateformes Orientées Composant Modernes

Microsoft

- Trois générations de plateformes composant
  - COM -> DCOM -> .Net (Inter-opérables)
  - Langages : C#, VB, C++, python, scheme, standard ML…
  - Une seule plateforme (Win32), plutôt vision locale à la machine
- Déclarations de :
  - Interfaces des Services : CTS : common type system
    - Agnostique au langage
  - ✓ Services : Binaires en CIL Common Interface Language
    - + ou – des DLL + des points d'entrée bien définis
  - ✓ Runtime : CRE Common Runtime Engine
    - Lie les différents composants entre eux, cycle de vie des composants…

Microsoft (2)

wikipedia



OSGI

- Plateforme orientée composant et centrée sur Java
  - Sous-jacent dans eclipse, mais applications diverses -> domotique
- Services :
  - Déclarés via des interfaces Java + configuration XML + Manifest.MF
    - Finalement neutre vis-à-vis d'une réalisation e.g. en C++
  - Notion de points d'extensions de la plateforme
    - Réaliser une certaine interface + se déclarer = plugin
- Composants / Bundle
  - Notion d'Activator
    - Install, start, stop, update
      - Gestion du cycle de vie du composant
    - Intanciation *lazy* des dépendances (résolution sur *start*), versioning

OSGI/Eclipse

- Services offerts par la plateforme eclipse
  - Registry
    - Les services actuellement disponibles sont inspectables par reflection
    - Les nouveaux services s'enregistrent et sont démarrés quand on les consulte
      - Intérêt du XML de configuration
    - Matérialise un annuaire
- Exemple : fournir un éditeur

+ réalisation de l'interface Java Colorisation, interception des commandes, complétion, erreurs…

```
<extension point="org.eclipse.ui.editors">
    <editor
      id="com.xyz.XMLEditor"
      name="Fancy XYZ XML editor"
      icon="./icons/XMLEditor.png"
      extensions="xml"
      class="com.xyz.XMLEditor"
      contributorClass="com.xyz.XMLEditorContributor"
      symbolicFontName="org.eclipse.jface.textfont"
      default="false">
    </editor>
 </extension>
```

J2EE

- Couche RMI : Remote Method Invocation
  - Réalisation d'un proxy distant général
    - Les arguments des opérations doivent être Serializable
  - ✓ Pour une interface Java annotée
    - Génération des Stub/skeleton + marshall/unmarshall par copie
    - Service de nommage
- J2EE Enterprise Edition
  - Basé sur RMI
  - Trois couches pour un composant (Bean)
    - Home (Logique applicative), Session (données locales), Entity (données stockées)
    - Déploiement dans divers contexte, e.g. serveur web ou d'applications
  - ✓ Pur Java, solution déclinante (lourdeur, interopérabilité)

Google

- Couche transport : Protobuf
  - Définition neutre du format des données
    - Génération pour divers langages des stub/skeleton
    - GO, C++, Java, Python, Ruby, …
  - Protocole de sérialisation efficace
    - Aussi utilisé pour stocker les données persistées
    - Gestion de versions des données
- Couche Services : GRPC
  - Déclaration de services via des signatures
    - Homogène avec la description protobuf + transport http

✓ Très populaire, eg. Netflix, la plupart des services Google, cisco, …solutions d'hébergement dans le cloud

Exemple Hello World gRPC

```
// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

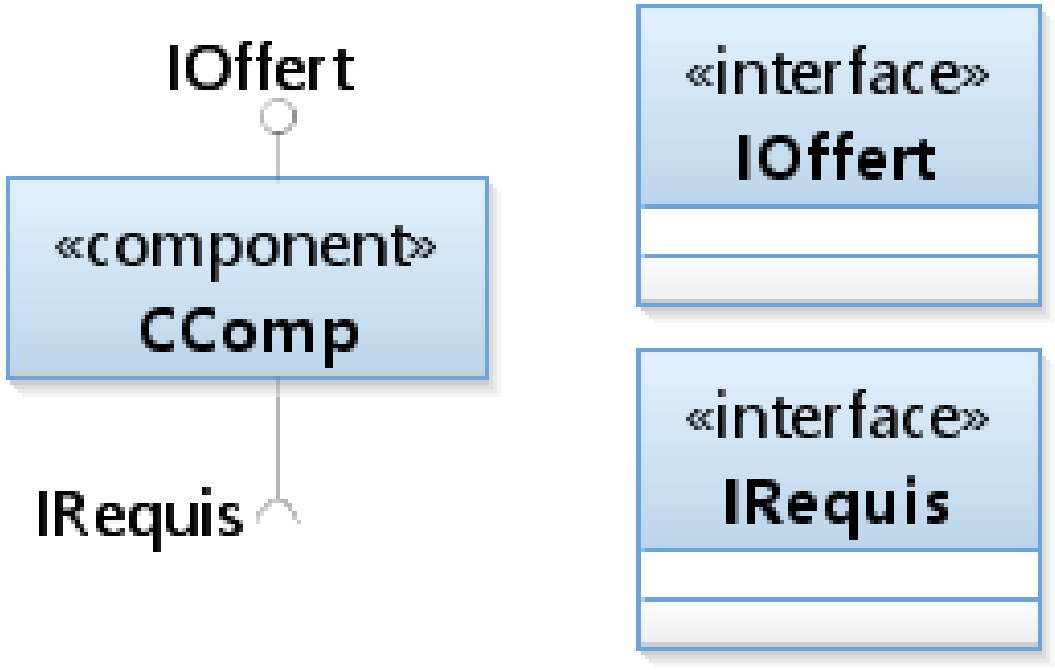
Composants -> Service

- L'orienté composant amène la notion de service
  - Service oriented-architecture SOA, Software-as-a-Service SaaS
  - Exposer du logiciel sous la forme de services dématérialisés
- Interfaces neutres pour les webservices
  - Certains standards comme WSDL, plus ou moins suivis
- Protocoles d'échanges assez bien balisés
  - http + SOAP + XML ou JSON
  - Requêtes http nature POST ou GET
  - Facilité d'interaction quelque soit le langage du client (shell ok)

Une conception détaillée en J2SE des composants

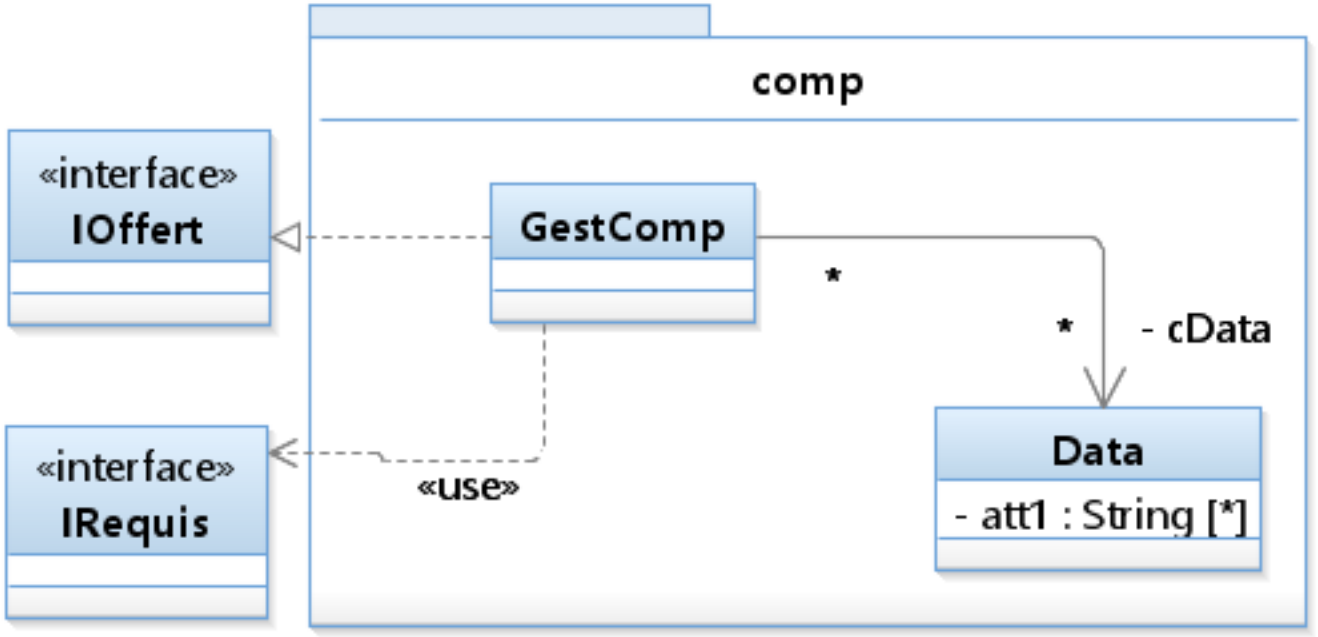
Des composants Java

- Objectifs :
  - matérialiser le discours, proposer une réalisation avec des outils connus
  - **Cohérence** entre la conception architecturale et détaillée
- Scope :
  - Intra-composant



Structure du Composant

- Cohérence :
  - Interfaces offertes :
    - implémentées par une classe du composant
  - Interfaces requises :
    - utilisées,
      - soit stockées dans des attributs
      - soit simplement invoquées, e.g. Singleton
    - /!\ multiplicité n'est pas visible sur le diagramme de composants



Mise en place

- Préparation de la génération de code
  - => structure du modèle importante
- Un composant **CComp** => un package **comp**
  - On trouve plus généralement api + api.impl, cf aussi modules Java 9+
- Un package pour loger les interfaces
  - Simpliste, principalement *ne pas placer les interfaces dans les package qui contiennent des implantations*
- Ne référer *que* aux interfaces quand on veut sortir du package/composant courant
  - Pas de dépendances structurelles entre package
  - Que des types simples ou des interfaces dans les signatures d'interfaces
  - Respect/Cohérence avec le diagramme de composants

Une façade pour le composant

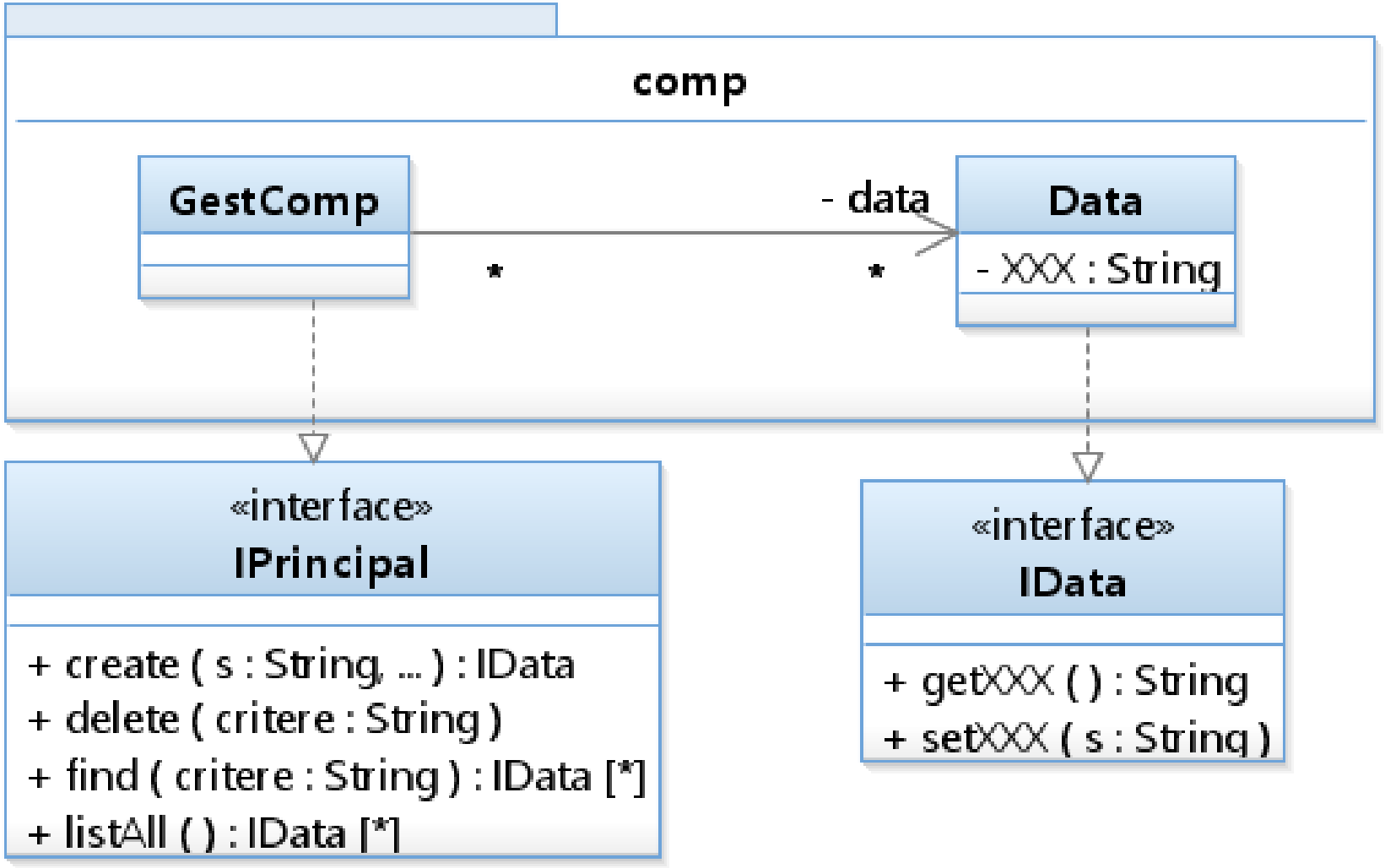
- Une classe qui réalise l'interface offerte principale du composant
  - Composant **CComp =>** classe **GestComp**
  - Une seule instance par instance de composant
  - Point d'entrée du composant
- Accède essentiellement à toutes les informations internes au composant
  - Cf la vision qu'a le système en analyse sur les données métier
  - Mais à réaliser maintenant :
    - Associations *
    - Map<clé, objet> pour représenter des index…
- Par défaut penser que GestComp réalise les interfaces offertes et utilise ou stocke les interfaces requises.

Composant de stockage

- Découpage des classes métier amène naturellement des composants de stockage
  - Responsabilité = gérer l'ensemble des exemplaires, leur état, leur statut…
  - Effet => GestComp connait * occurrences de Data
- API CRUD : Create, Replace, Update, Delete
  - Deux options principales :
    - API par sous interfaces, orienté objet mais respectant les contraintes du composant
    - API orienté composant pur, utilisant beaucoup d'identifiants

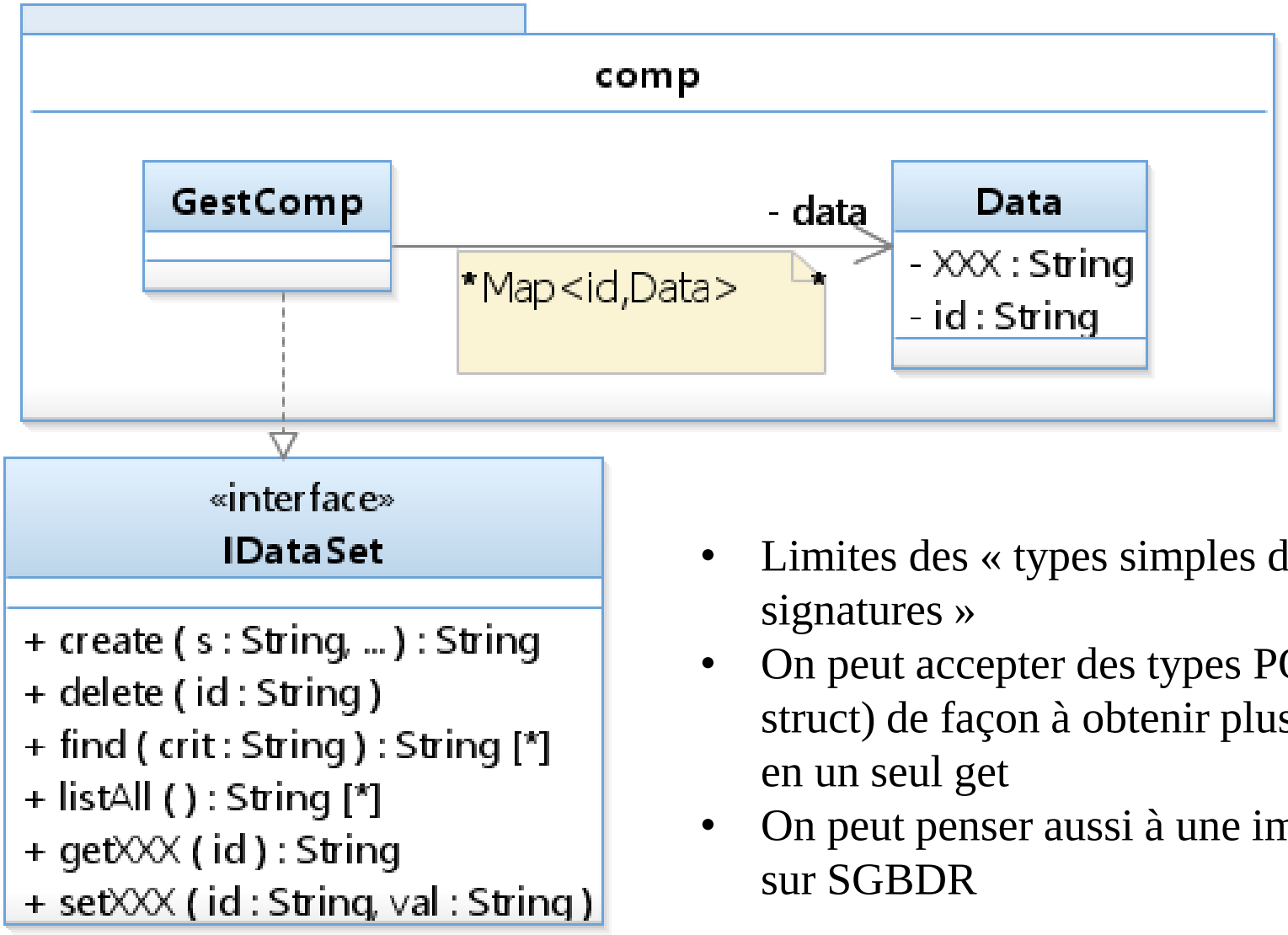
Version OO

CComp offre deux interfaces : IPrincipal et IData



Version Composant « pur » par ID

On évite d'exposer des interfaces nouvellement crées / internes



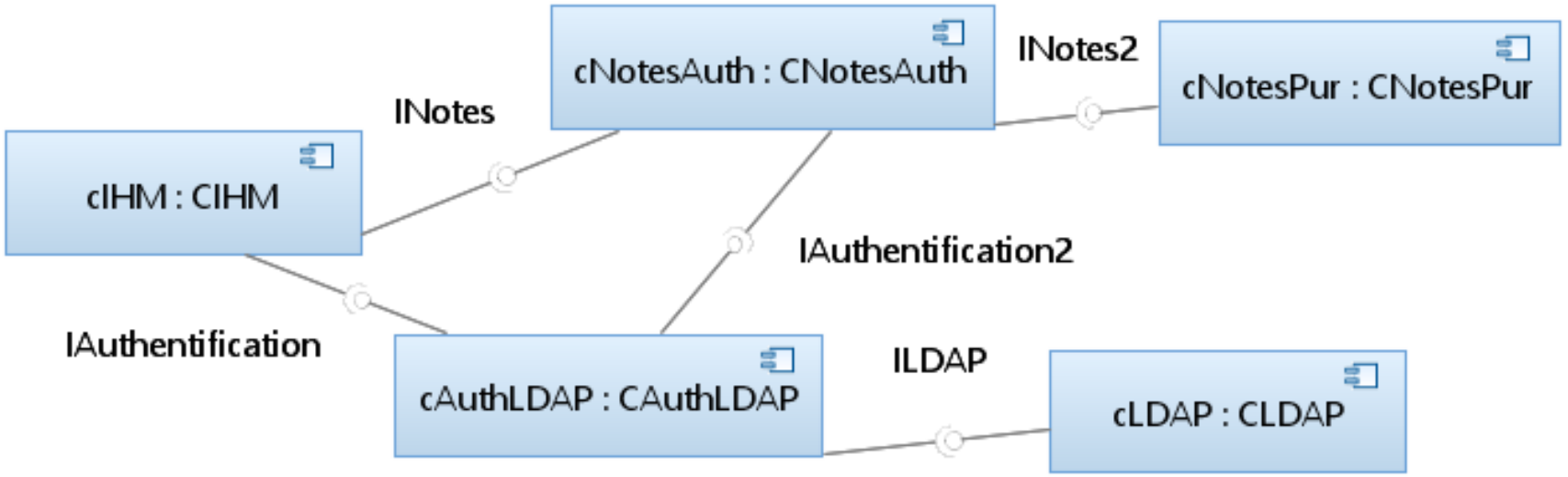
- Limites des « types simples dans les signatures »
- On peut accepter des types POD (pures struct) de façon à obtenir plusieurs attributs en un seul get
- On peut penser aussi à une implémentation sur SGBDR

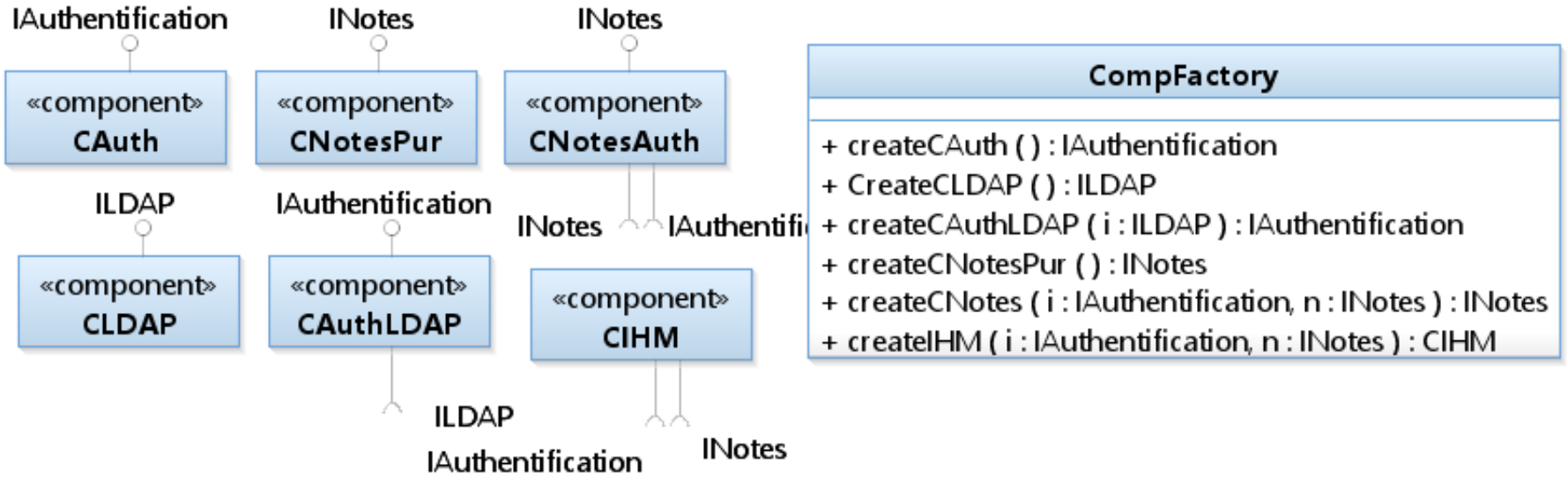
Instanciation et Configuration

Instancier ces classes en respectant le modèle de composants ?

- Application d'un DP Factory
  - Une classe chargée de construire des abstractions sans exposer les implémentations
  - createCarre(x,y,a):IForme
- Choix basique dans l'UE :
  - une seule classe de factory static isolée dans son package
  - Grosses dépendances sur tout le reste OK
- Opérations de la factory :
  - Création de composants, rendus via un typage sur interface offerte
  - Les dépendances des composants sont passés si possible à la construction
    - Sous la forme d'interfaces !
  - ✓ Si possible on ordonne les créations => DAG de construction
  - ✓ S'il existe des cycles
    - => il existe des setter, provider, singleton pour fermer le cycle après instanciation

Exemple


cNotesAuth : CNotesAuth
INotes2
cNotesPur : CNotesPur
INotes
cIHM : CIHM
IAuthentification2
IAuthentification
cAuthLDAP : CAuthLDAP
ILDAP
cLDAP : CLDAP


IAuthentification
INotes
INotes
«component»
CAuth
«component»
CNotesPur
«component»
CNotesAuth
ILDAP
IAuthentification
INotes
IAuthentifi
«component»
CLDAP
«component»
CAuthLDAP
«component»
CIHM
ILDAP
IAuthentification
INotes
CompFactory
+ createCAuth ( ) : IAuthentification
+ CreateCLDAP ( ) : ILDAP
+ createCAuthLDAP (i : ILDAP) : IAuthentification
+ createCNotesPur ( ) : INotes
+ createCNotes (i : IAuthentification, n : INotes) : INotes
+ createIHM (i : IAuthentification, n : INotes) : CIHM

Dependency Injection

- Au-delà de Factory
  - Pattern général : injection de dépendance
  - Le client déclare ses dépendances, un mécanisme externe se charge de les lui fournir
- Exemple Google Guice
  - Intégration dans Java d'un framework de DI
  - Généralisation des stratégies :
    - @Inject IRequis req;
  - ✓ Définition séparée de « modules »
    - Ensemble de bindings entre une interface et une réalisation concrète
    - Externe au code qu'on configure
  - ✓ Facilité de mise en place des tests d'intégration par exemple
  - ✓ Reconfigurations aisée, frameworks extensibles e.g. XText…

Guice exemple

```
public class BillingModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(TransactionLog.class).to(DatabaseTransactionLog.class);
    bind(CreditCardProcessor.class).to(PaypalCreditCardProcessor.class);
    bind(BillingService.class).to(RealBillingService.class);
  }
}
```

```
public class RealBillingService implements BillingService {
  private final CreditCardProcessor processor;
  private final TransactionLog transactionLog;

  @Inject
  public RealBillingService(CreditCardProcessor processor,
      TransactionLog transactionLog) {
    this.processor = processor;
    this.transactionLog = transactionLog;
  }
```

Conception Détaillée : Bilan

Une conception « fine »

- Objectifs :
  - Lever les points d'ombre sur la façon de réaliser les traitements,
  - organiser le code, choisir des structures de données, introduire des DP…
  - Cf Janvier 2018 pour des exemples niveau examen, Sudoku pour la structure du projet/code
- Choix techniques guidés par les API
  - E.g. index/map si on fournit une interface par ID
- Mais savoir s'arrêter !
  - Quand c'est clair, coder, puis resynchroniser les diagrammes sur le code (round-trip engineering)
- Un choix possible abordé dans l'UE mais nombreuses façons de réaliser des composants
  - Dépendances fonctionnelles *seulement* entre parties du système