Du découplage à l'assemblage non-anticipé de composants

Conception et mise en œuvre du langage à composants Scl

Luc Fabresse

Université Montpellier 2

Laboratoire d'Informatique de Robotique et de
Micro-électronique de Montpellier

Soutenance de thèse

10 Décembre 2007

Plan

Plan

Facteurs problématiques en génie logiciel

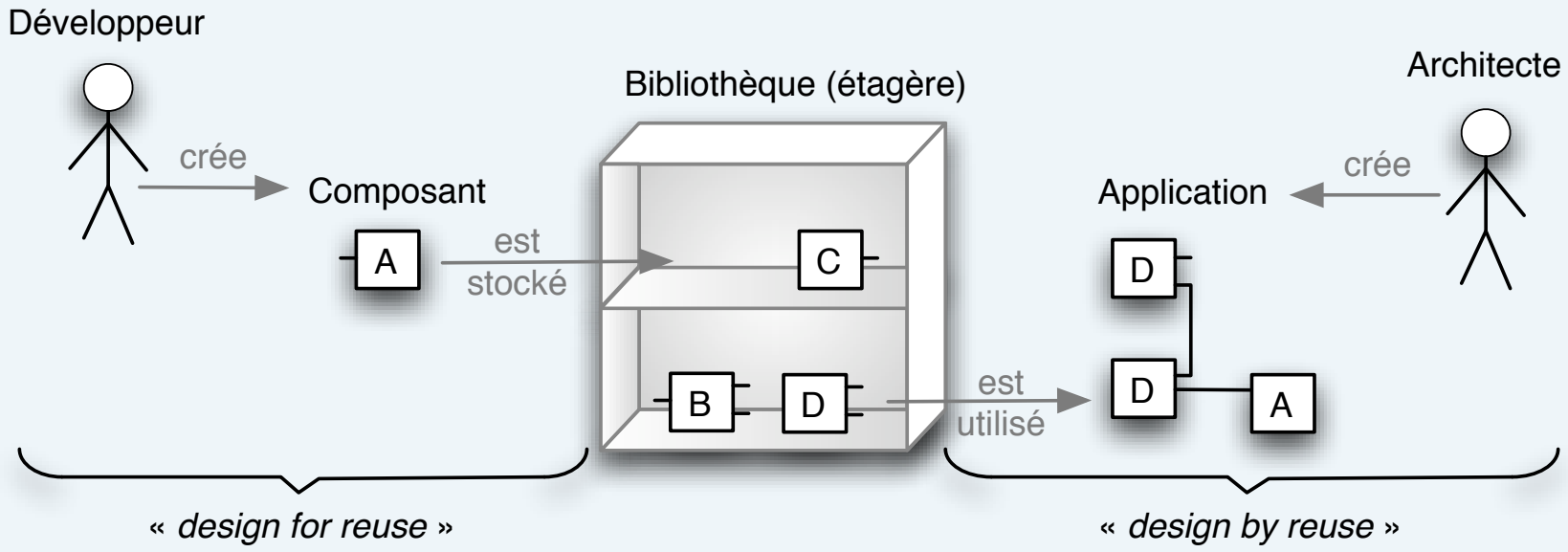
- Coûts et temps (développement, maintenance, tests, évolutions)
- Taille des logiciels
- Évolution (correction de bugs, nouvelles fonctionnalités)
- Réalisation de tests (unitaires, intégrations, couverture, non-régression)
- Inter-opérabilité (plateformes, langages)

Le paradigme « composant »

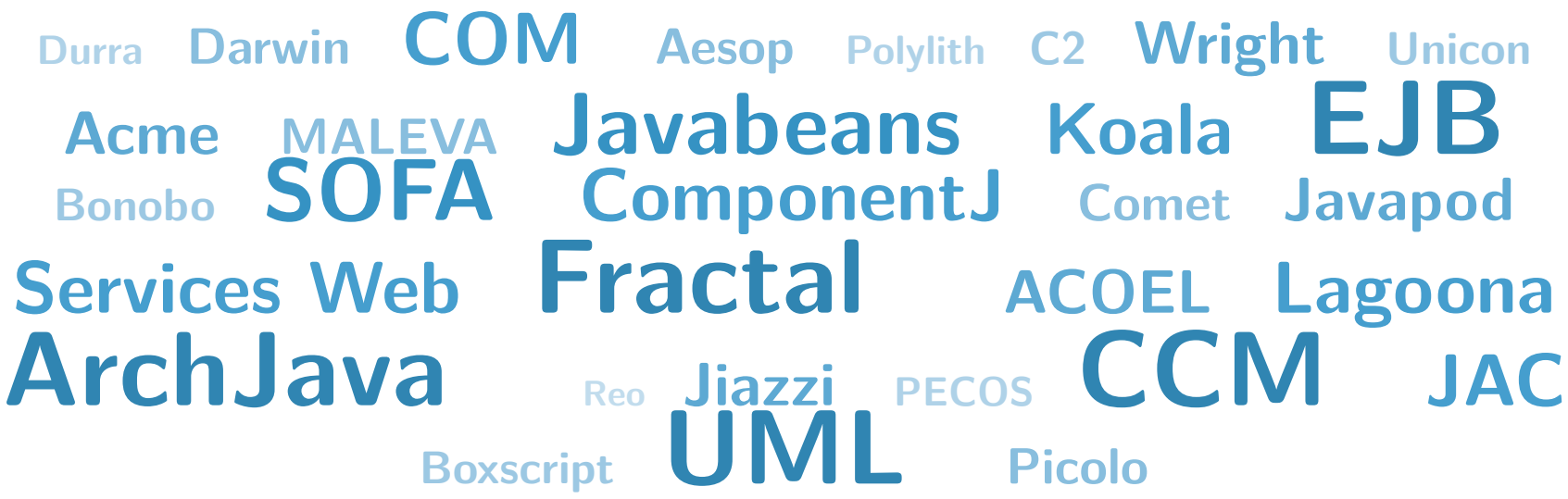
Objectifs

- Mieux réutiliser
- Mieux décrire l'architecture des logiciels (les interactions)

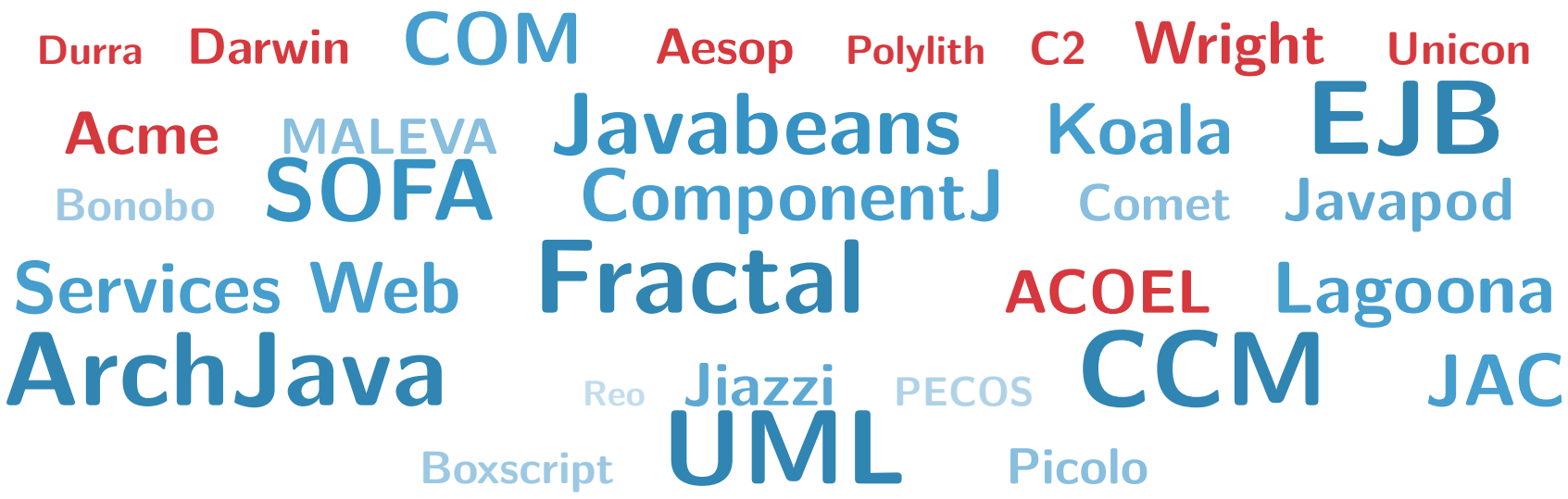
La programmation par composants (PPC)



Classification par grandes familles



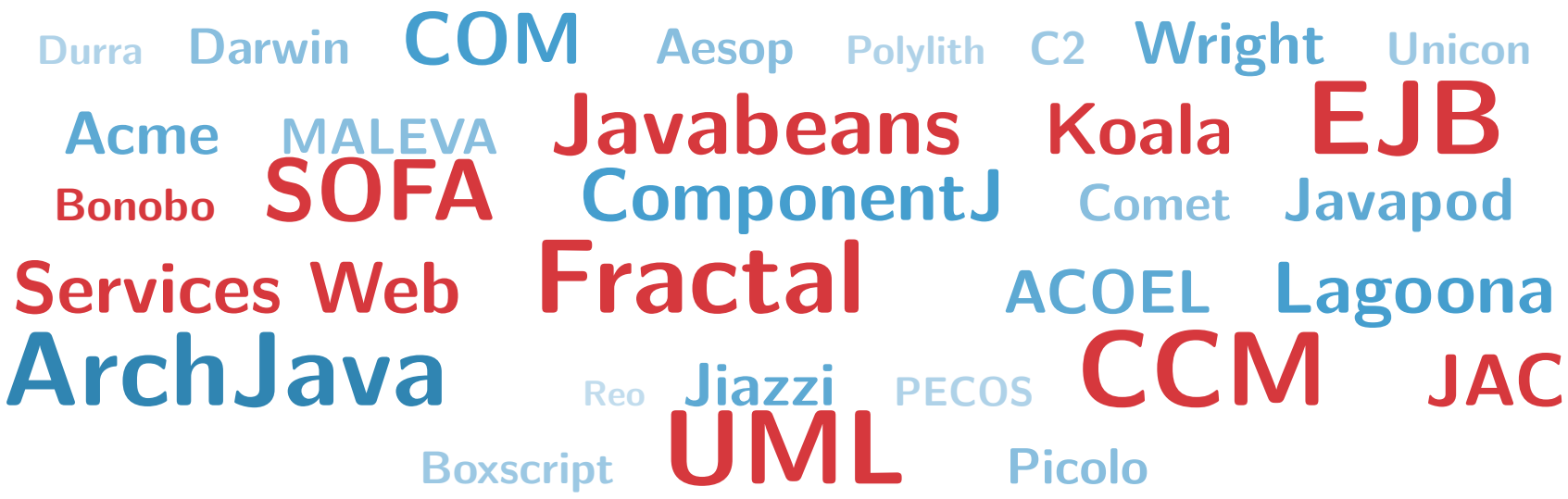
Classification par grandes familles



Un classification globalement consensuelle

- Langages de description d'architecture (ADLs)

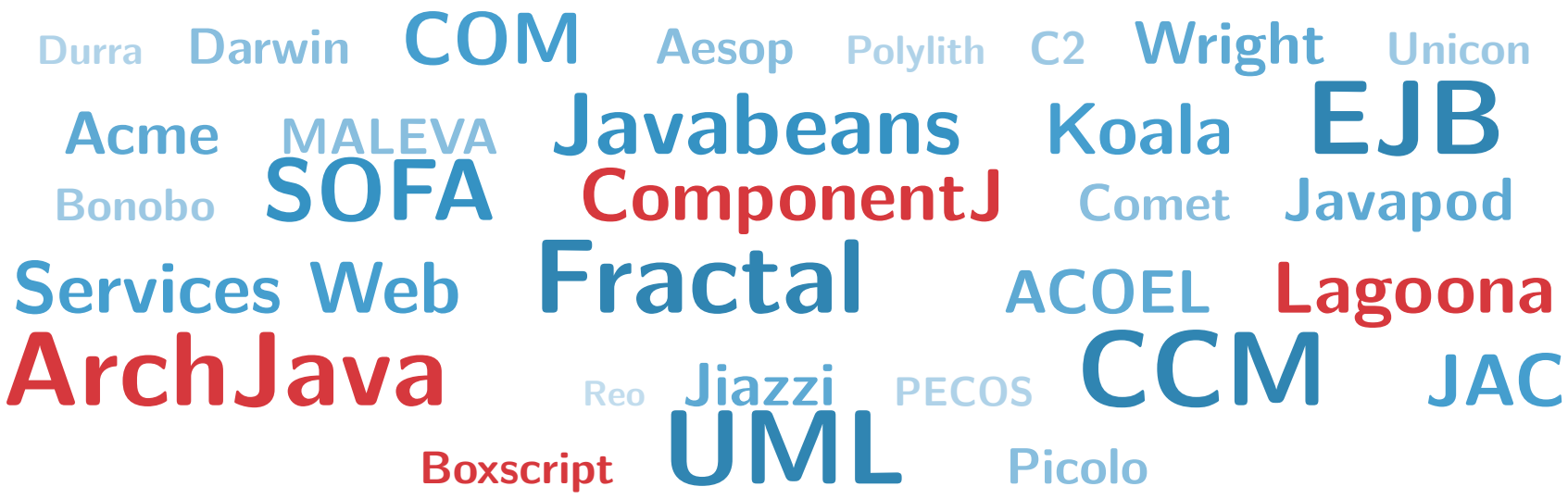
Classification par grandes familles



Un classification globalement consensuelle

- Langages de description d'architecture (ADLs)
- Modèles de composants

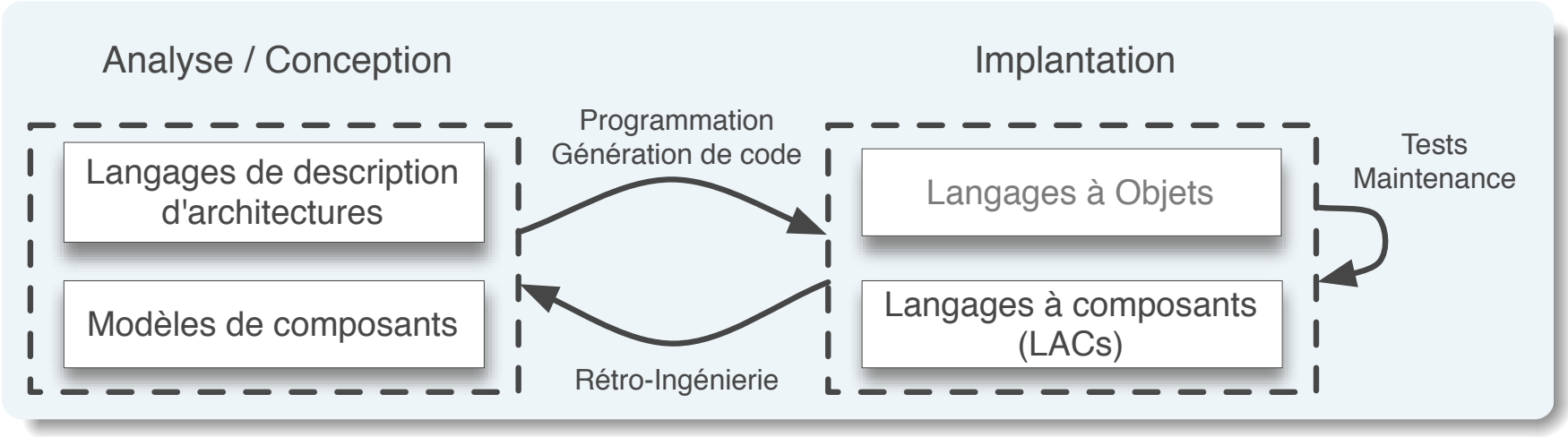
Classification par grandes familles



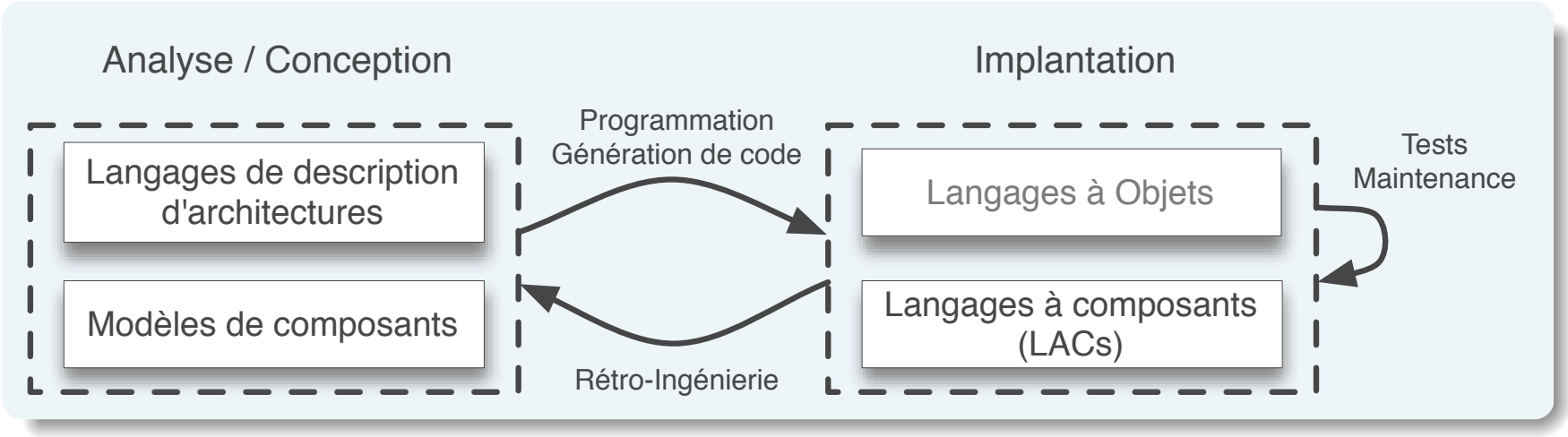
Un classification globalement consensuelle

- Langages de description d'architecture (ADLs)
- Modèles de composants
- Langages à composants (LACs)

De l'utilité des langages à composants (LACs)



De l'utilité des langages à composants (LACs)



Programmation par composants : 4 techniques

- Langage de programmation + conventions (Javabeans)
- Extension d'un *framework* Objet (EJB)
- Transformation de modèles (Fractal → Julia)
- Langages à composants (ArchJava)

Notre problématique

Constats

- Prise en compte partielle du découplage et de la non-anticipation
- Diversité des concepts
- Mise en pratique difficile de la PPC (peu de LACs)
- Les LACs sont généralement des extensions de langages à objets
  - Sous-spécifiés (ex : Auto-référence)
  - Non-uniformes (ex : ArchJava Objet + Composants)

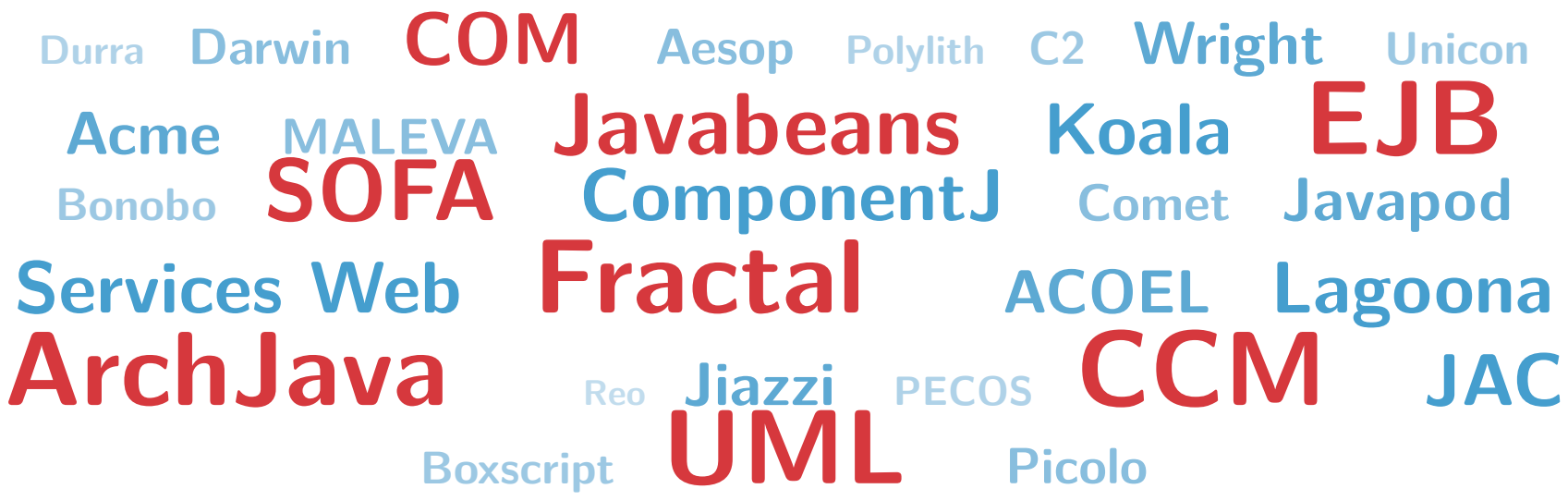
Notre objectif

Réaliser un langage de programmation

- Intégrant les concepts et mécanismes nécessaires et suffisants pour faire de la PPC (Minimal)
- Prenant en compte le découplage et la non-anticipation
- N'entretenant pas la confusion objets/composants (Uniforme)
- Offrant des mécanismes de haut niveau (Simple)

Plan

État de l'art : LACs et problèmes existants



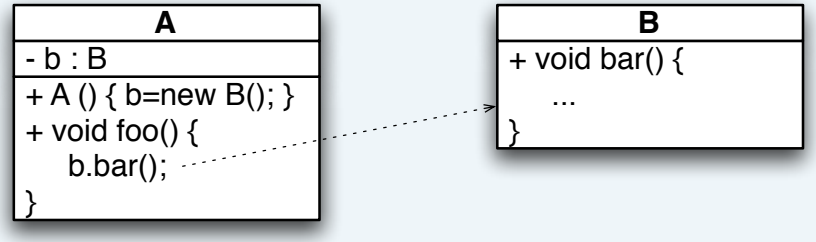
Plan

1. Le problème du « découplage »
2. Le problème de la « non-anticipation »
3. Diversité des concepts
4. Sous-spécifications et non-uniformité des LACs

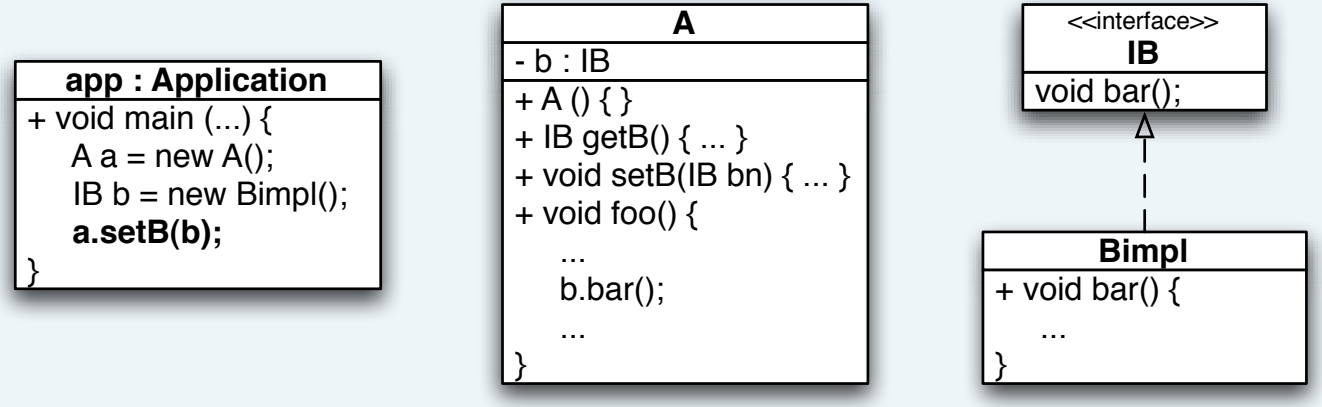
Plan

Notion de « (dé)couplage » en POO

Exemple de couplage

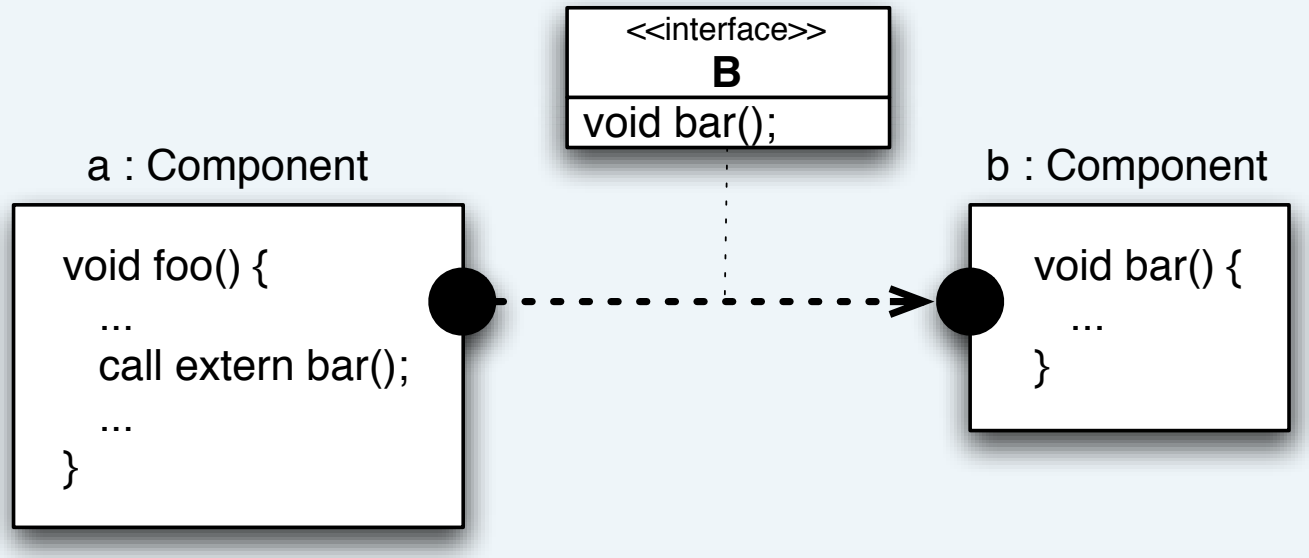


Exemple de découplage



En PPC : une volonté de « découplage »

Exemple



Le découplage dans les approches à composants

Niveaux de découplage

- ☹ Un client peut référencer directement un fournisseur
- 😟 Un client ne référence pas directement un fournisseur
- ☺ Un client ne référence pas un fournisseur qui peut être changé dynamiquement

Comparatif

	Découplage
LOO (Java)	☹
D/COM/+	☹
Javabeans	☺
EJB	☹
CCM	😟
Fractal	☺
SOFA	☺
ArchJava	😟

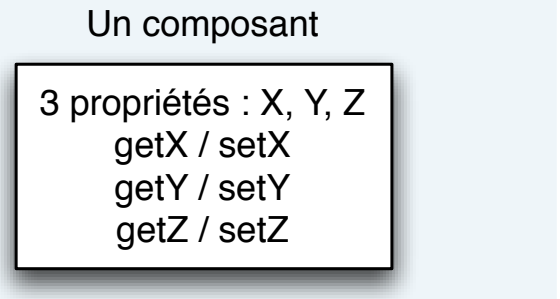
Plan

Notion de « non-anticipation »

L'idée de « Non-anticipation »

Le programmeur d'un composant ne doit pas intégrer de code spécifique dans un composant afin de permettre son assemblage

Non-anticipation



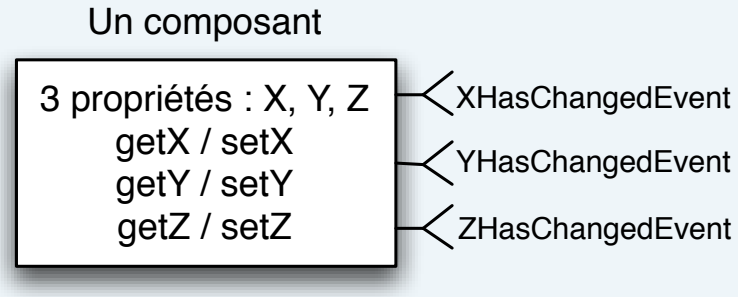
Notion de « non-anticipation »

L'idée de « Non-anticipation »

Le programmeur d'un composant ne doit pas intégrer de code spécifique dans un composant afin de permettre son assemblage

Non-anticipation

Anticipation



La non-anticipation dans les approches à composants

Niveaux de non-anticipation

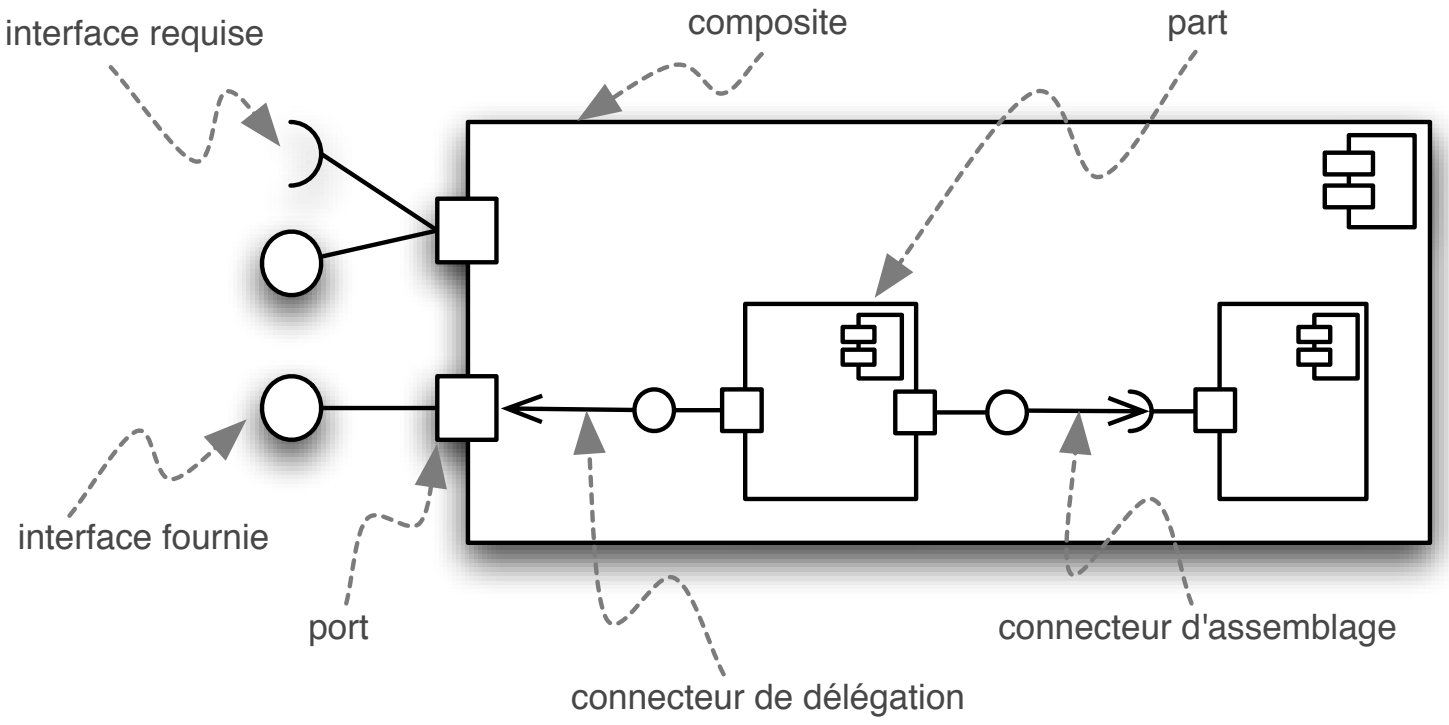
- ☹ Du code est nécessaire pour les connexions de type requis–fournis
- ☹ Du code est nécessaire pour les connexions événementielles
- ☺ Aucun code nécessaire pour autoriser ces deux types de connexions

Comparatif

	Découplage	Non-anticipation
Java	☹	☹
D/COM/+	☹	☹
Javabeans	☺	☹
EJB	☹	☹
CCM	☹	☹
Fractal	☺	☹
SOFA	☺	☹
ArchJava	☹	☹

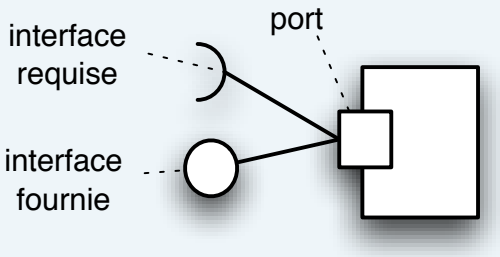
Plan

Présentation des principaux concepts à travers la notation graphique d'UML 2.0



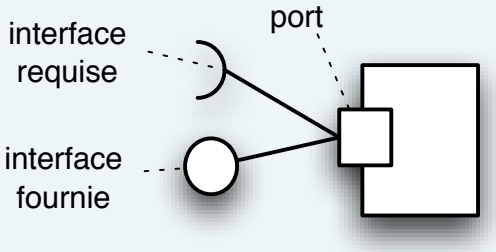
Les concepts de « port » et d'« interface »

UML

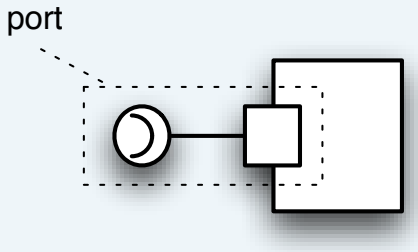


Les concepts de « port » et d'« interface »

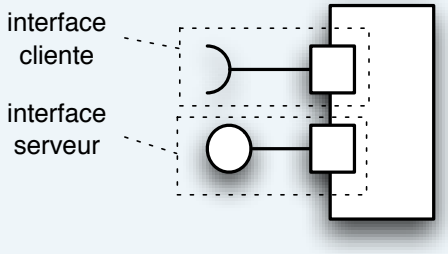
UML



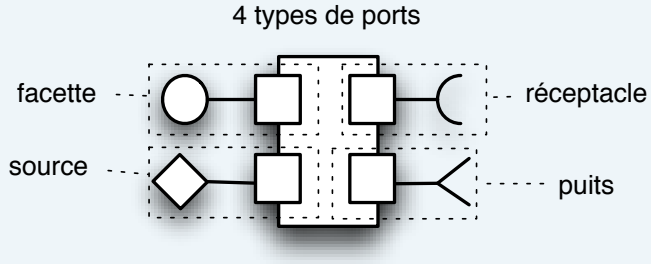
ArchJava



Fractal



CCM

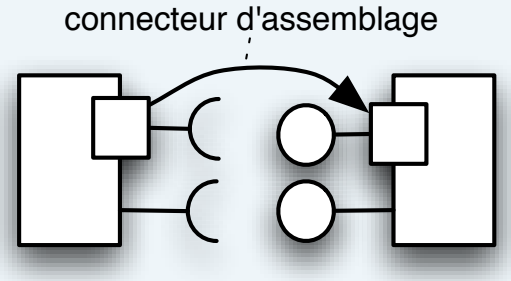


Diversité des concepts

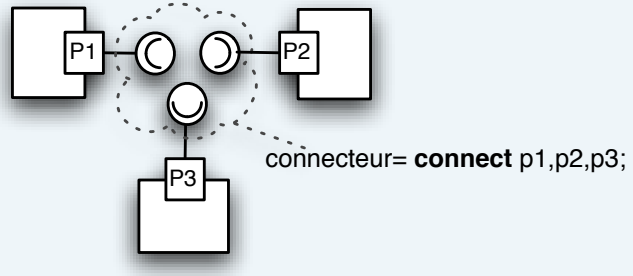
- Communications unidirectionnelles / bidirectionnelles
- Nombre d'interfaces par port

Les concepts de « connexion » et de « connecteurs »

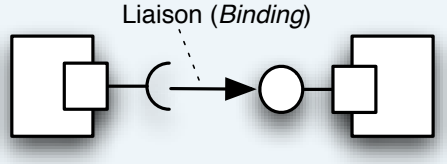
UML



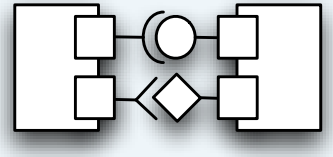
ArchJava



Fractal



CCM



Diversité des concepts

- Que connecte-t-on : les ports et/ou les interfaces ?
- Les connexions sont-elles binaires et/ou n-aires ?
- Les connexions sont-elles réifiées (connecteurs) ?

Plan

Sous-spécifications et non-uniformité dans les LACs

Présentation de deux problèmes

A L'échange de références entre objets en Julia

B La non-uniformité d'ArchJava

Sous-spécifications

- Comment s'effectue le passage d'arguments ?
- Comment intégrer les objets et les composants ?
- ...

Sous-spécifications et non-uniformité dans les LACs

Présentation de deux problèmes

A **L'échange de références entre objets en Julia**

B La non-uniformité d'ArchJava

Sous-spécifications

- **Comment s'effectue le passage d'arguments ?**
- Comment intégrer les objets et les composants ?
- ...

Sous-spécifications et non-uniformité dans les LACs

Présentation de deux problèmes

A L'échange de références entre objets en Julia

B **La non-uniformité d'ArchJava**

Sous-spécifications

- Comment s'effectue le passage d'arguments ?
- **Comment intégrer les objets et les composants ?**
- ...

Problème de l'échange de références entre objets

Illustration en Julia (1)

Présentation de Julia : implémentation de référence de Fractal

- un *framework* Java
- des objets représentent
  - les interfaces des composants
  - l'implémentation des composants
- le modèle d'exécution de Java

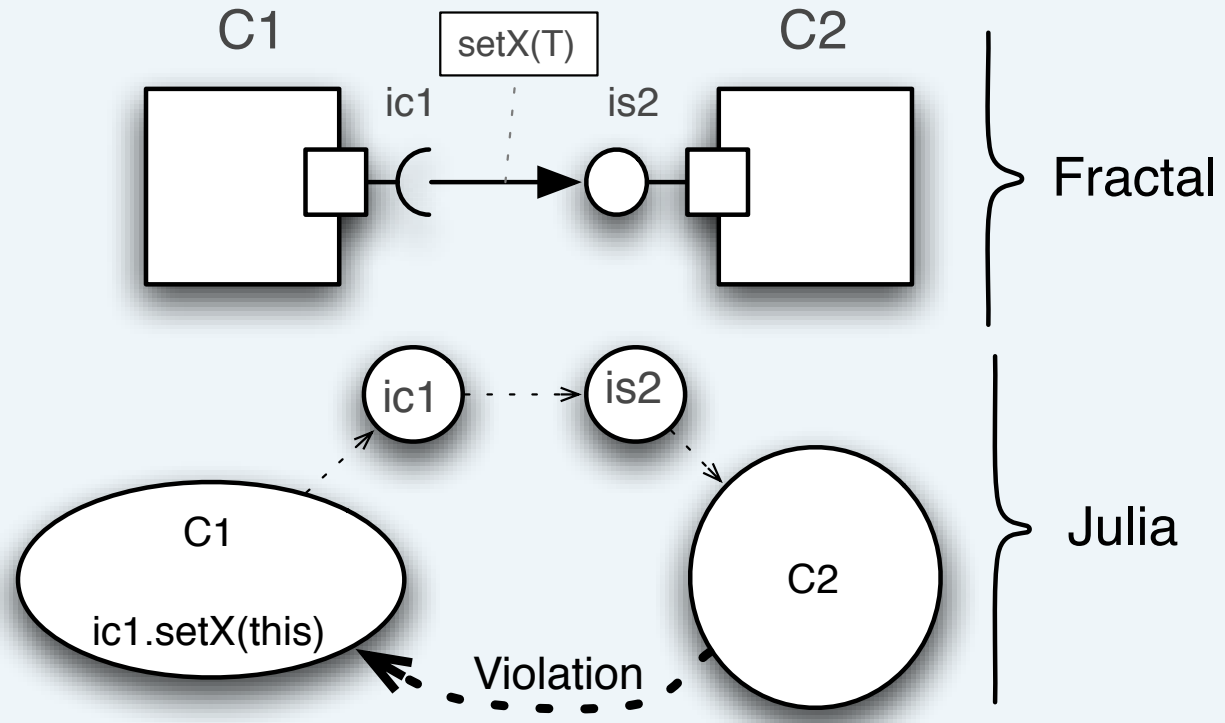
Définition : Intégrité des communications

Toute communication entre composants s'effectue via une connexion

Problème de l'échange de références entre objets

Illustration en Julia (2)

Violation de l'intégrité des communications en Julia [LCL06]



Problème de l'échange de références entre objets

La solution d'ArchJava

Présentation d'ArchJava : une extension du langage Java

- Notion de « classe de composants »
- Notion de « Port »
- La primitive `connect` (`connect` $p_1, p_2, \ldots, p_n$)
- La primitive `glue`
- ...

Contrôle de l'intégrité des communications en ArchJava

- basé sur le système d'annotation de types « *AliasJava* »
- effectué lors de la compilation

Problème de la non-uniformité

Illustration en ArchJava avec un programme « Hello World »

Helloer.archjava

```
component class Helloer {
   public port hello {
      provides String
         sayHello(String name);
   }
   public port printer {
      requires void print(String s);
   }

   String sayHello(String name) {
      printer.print("Hello, " + name);
   }
}
```

StdInputOutput.archjava

```
component class StdInputOutput {
   public port stdout {
      provides void print(String s);
   }
   void print(String s) {...}
}
```

Exemple de connexion

```
Helloer h = new Helloer();
StdInputOuput p = new StdInputOutput();
connect h.print, p.stdout;
h.hello.sayhello("luc");
```

Problème de la non-uniformité

Illustration en ArchJava

La cohabitation objet/composant en ArchJava

- Un composant est connectable et ne peut être passé en argument
- Un objet n'est pas connectable mais peut être passé en argument

Peut-on imaginer un monde uniforme ?

- Comment intégrer les objets et les composants ?
- Comment représenter les objets de base ?
- Que signifierait passer un composant en argument ?

Synthèse comparative

	Julia	ArchJava
Passage d'arguments	☹	☹
Uniformité	☹	☹
Auto-référence	☹	☹
Découplage	☺	☹
Non-anticipation	☹	☹

Plan

Nos objectifs pour Scl

Que doit permettre SCL ?

- Programmer des composants (Découplage)
- Assembler des composants (Non-anticipation)

Démarche

- Conception *bottom-up* (Minimalité)
- Intégration uniforme des concepts et des mécanismes (Uniformité)

Les problématiques abordées

Les choix pour SCL

- Faut-il des descripteurs de composants ? (Choix 1-3)
- Faut-il des ports et/ou des interfaces ? (Choix 4-6)
- Comment intégrer uniformément les types de base et les composants ? (Choix 7-8)
- Faut-il des connexions simples et/ou multiples ? (Choix 9-10)
- Qu'est-ce qu'« invoquer un service » ? (Choix 11-14)
- Faut-il réifier les connexions ? (Choix 15-18)
- Faut-il des composites ? (Choix 18-20)

Des ports et/ou interfaces et sous quelle forme ?

Pourquoi séparer les concepts de port et d'interface ?

- Les ports supportent les connexions et les invocations de services
- Les interfaces définissent des contrats d'assemblage/d'utilisation

Des ports unidirectionnels (Choix 4)

- Respect de la non-anticipation
- Suffisamment expressifs
- ☹ Imposer l'unicité du composant fournisseur et client

Des ports et/ou interfaces et sous quelle forme ?

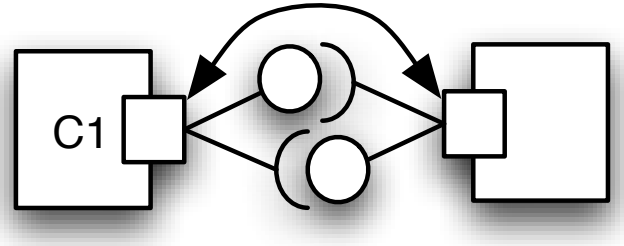
Pourquoi séparer les concepts de port et d'interface ?

- Les ports supportent les connexions et les invocations de services
- Les interfaces définissent des contrats d'assemblage/d'utilisation

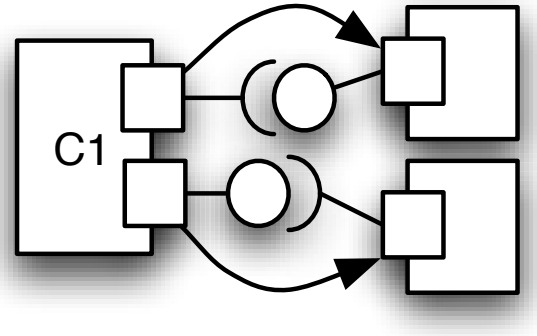
Des ports unidirectionnels (Choix 4)

- Respect de la non-anticipation
- Suffisamment expressifs
- ☹ Imposer l'unicité du composant fournisseur et client

Bidirectionnel



Unidirectionnel



Quelle forme pour les interfaces ?

Une interface est associée à un port (Choix 5)

- Les ports sont les points d'accès du composant
- Simplicité (une seule interface par port)

Interface = {signature de service} (Choix 6)

- ☹ Vérification syntaxique des assemblages uniquement
- Relation de sous-typage entre interfaces est l'inclusion ensembliste (Non-anticipation)

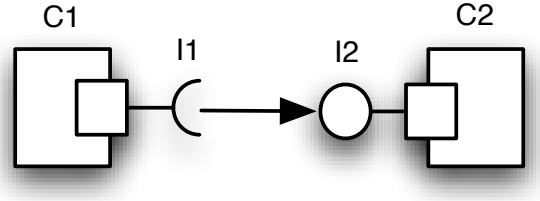
Quelle forme pour les interfaces ?

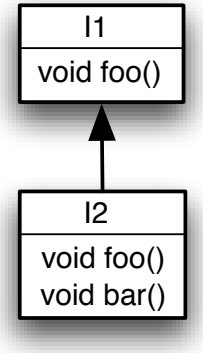
Une interface est associée à un port (Choix 5)

- Les ports sont les points d'accès du composant
- Simplicité (une seule interface par port)

Interface = {signature de service} (Choix 6)

- ☹ Vérification syntaxique des assemblages uniquement
- Relation de sous-typage entre interfaces est l'inclusion ensembliste (Non-anticipation)





Que signifie : « connecter des composants » ?

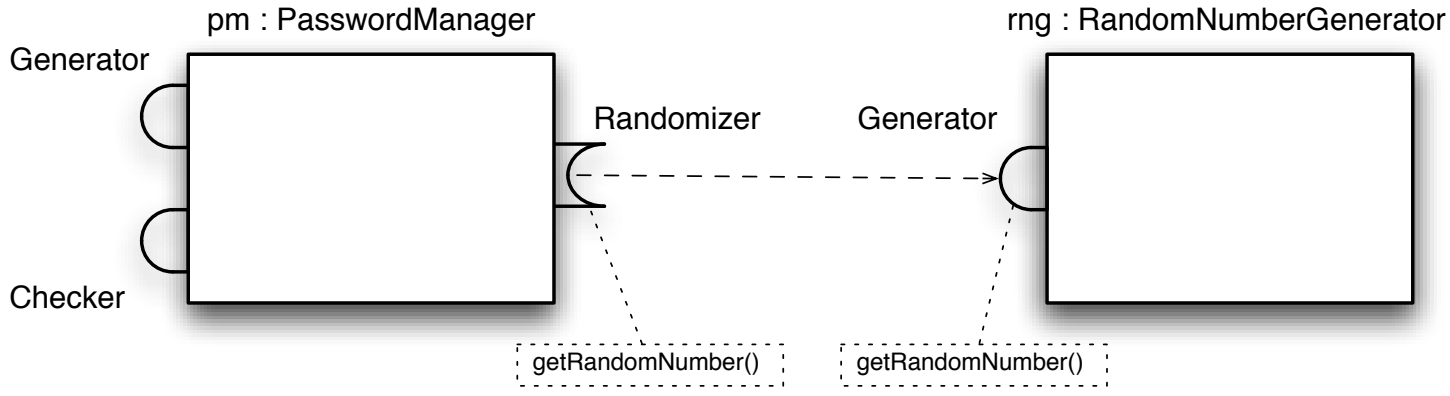
Liaison binaire : un port requis $\longrightarrow$ un port fourni (Choix 9)

- Suffisamment expressif
- Moins d'ambiguïtés (ex : ArchJava)

Que signifie : « connecter des composants » ?

Liaison binaire : un port requis ⟶ un port fourni (Choix 9)

- Suffisamment expressif
- Moins d'ambiguïtés (ex : ArchJava)



La primitive : **bind** <port requis> **to** <port fourni>

```
pm := new PasswordManager
rng := new RandomNumberGenerator
bind pm.Randomizer to rng.Generator
```

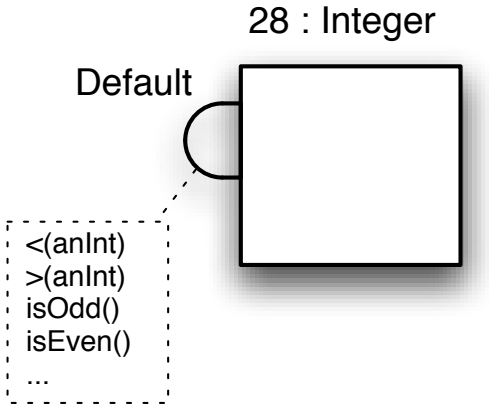
Comment intégrer les types de base ?

Les types de base (ex : entier) sont des composants (Choix 7)

Uniformité

Tout composant possède un port fourni par défaut (Choix 8)

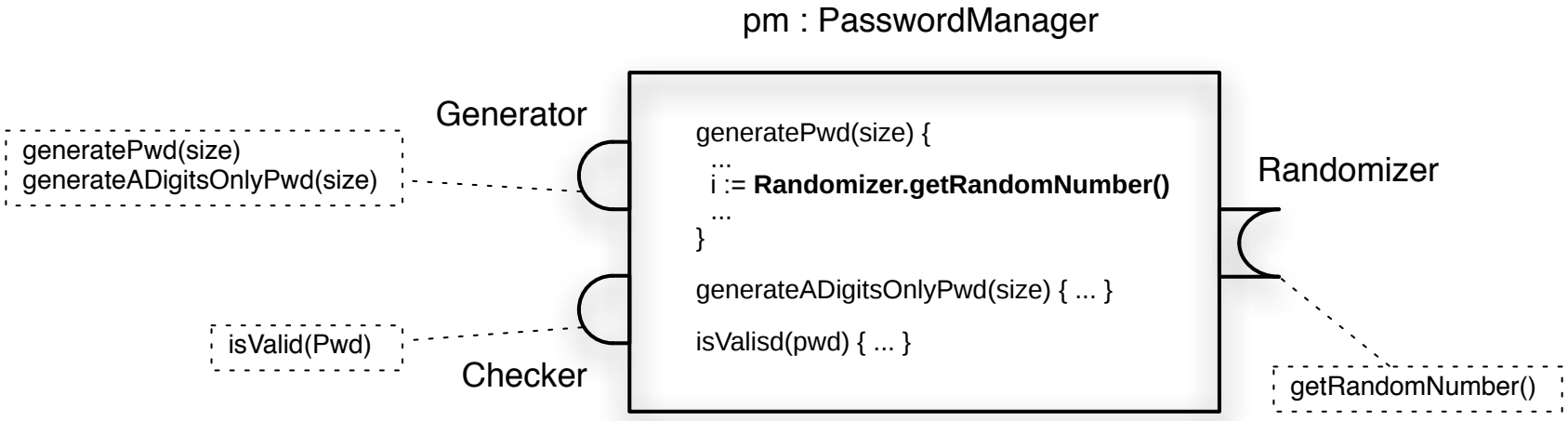
- Uniformisation de la structure minimale d'un composant
- Un objet possédant un port fourni par défaut = un composant



Comment invoquer un service ?

Toujours à travers un port (Choix 11)

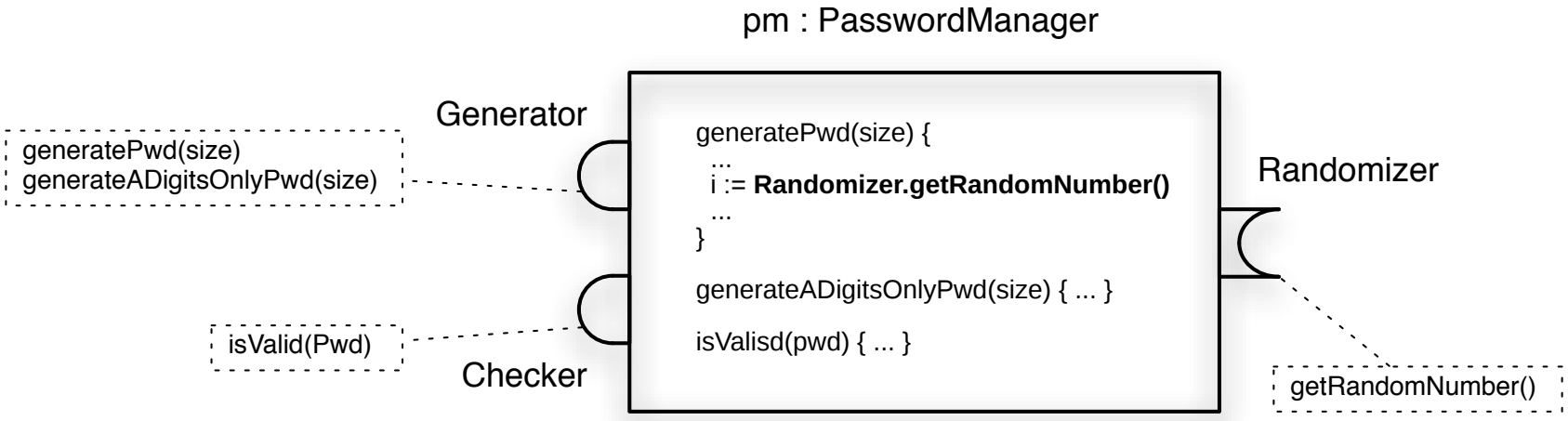
Communications via les ports uniquement (découplage, non-anticipation)



Comment invoquer un service ?

Toujours à travers un port (Choix 11)

Communications via les ports uniquement (découplage, non-anticipation)

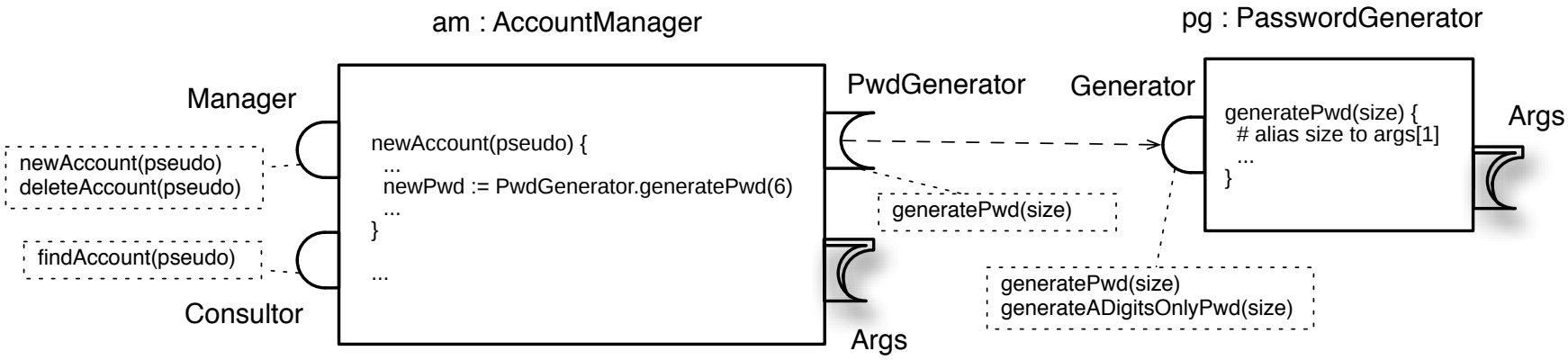


Les arguments sont des références sur des ports (Choix 13)

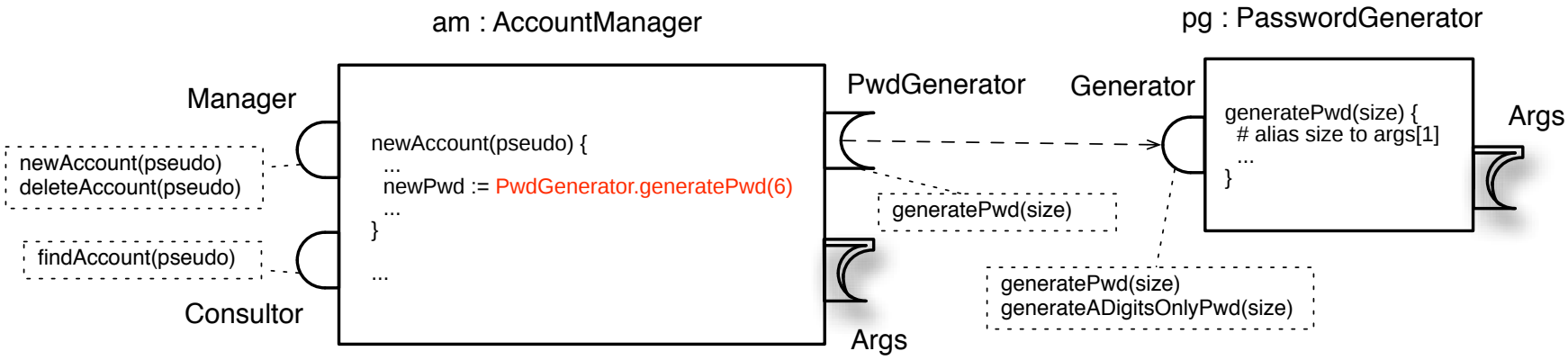
Référencer directement un composant :

- Introduirait du couplage
- Imposerait de contrôler les échanges de références

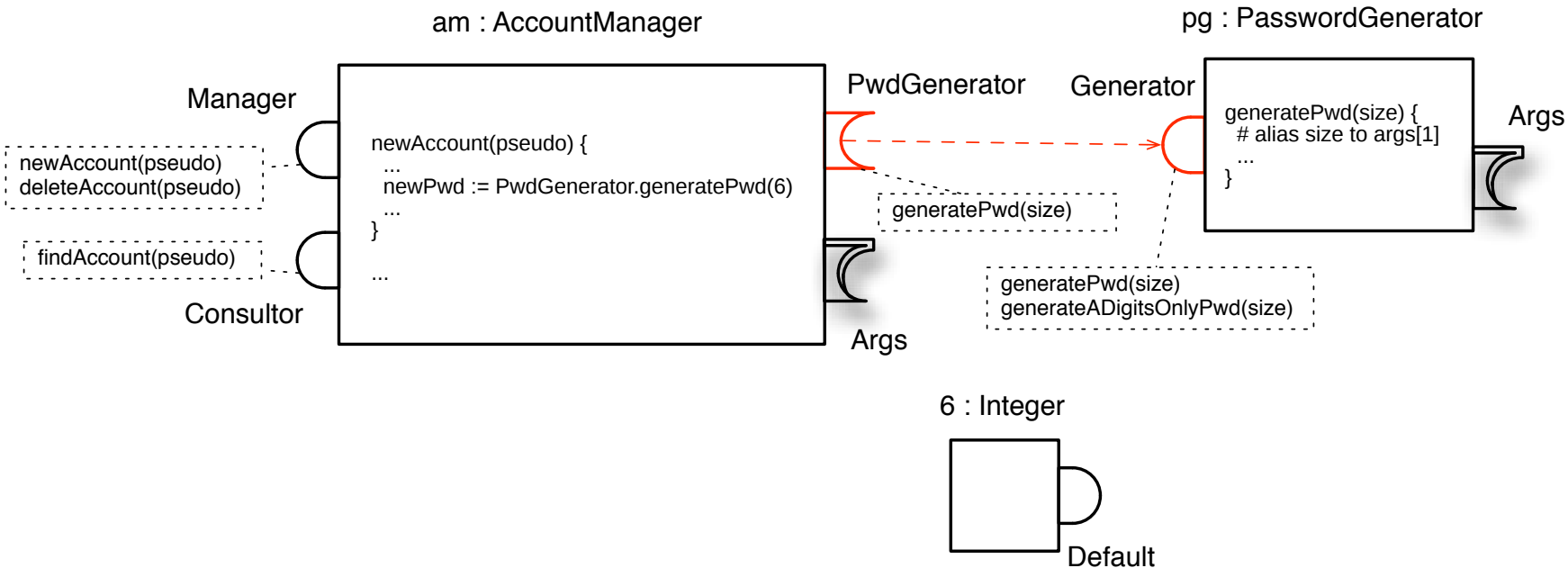
Passage d'arguments par connexion et déconnexion automatique de ports (Choix 14)



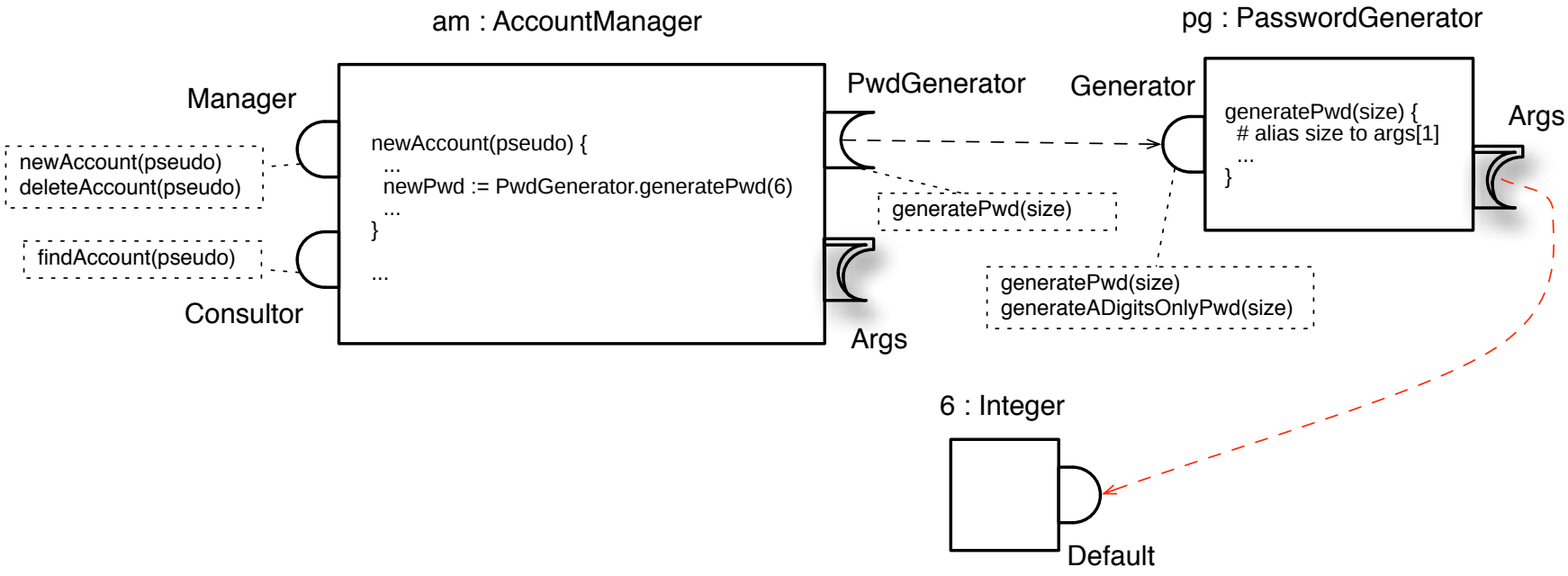
Passage d'arguments par connexion et déconnexion automatique de ports (Choix 14)



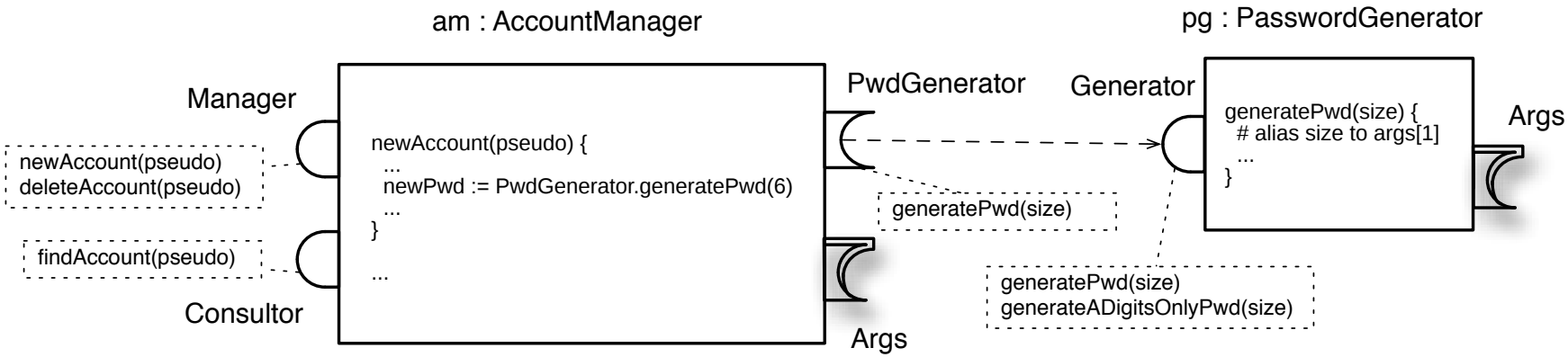
Passage d'arguments par connexion et déconnexion automatique de ports (Choix 14)



Passage d'arguments par connexion et déconnexion automatique de ports (Choix 14)



Passage d'arguments par connexion et déconnexion automatique de ports (Choix 14)



Faut-il réifier les connexions ?

Oui : nécessité des connecteurs (Choix 15)

Séparer le code métier et le code de l'interaction (Séparation des préoccupations)

Comment : un connecteur est un composant (Choix 16)

Uniformité

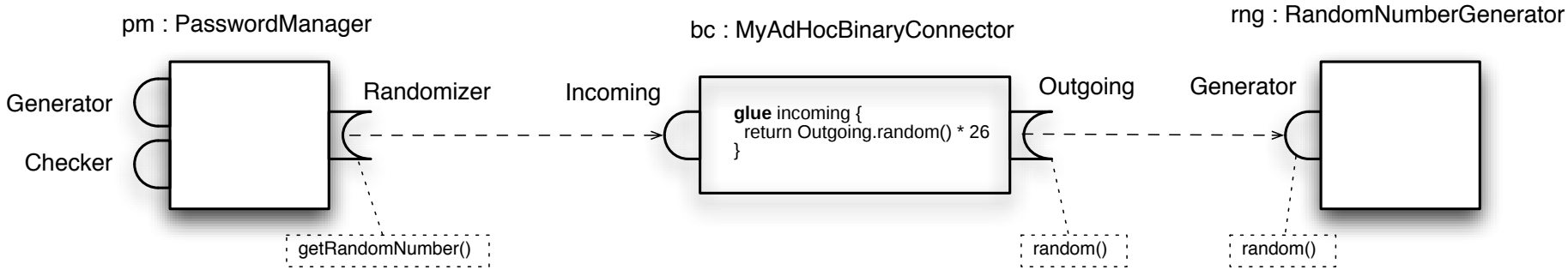
Problème

- Traiter les adaptations des invocations de service
- Produire des connecteurs réutilisables et paramétrables

Solution

Associer du « code glue » à un port fourni (Choix 17)

- Service particulier associé à un port fourni
- Exécuté en cas d'absence du service demandé
- Accès à l'invocation de service courante (pseudo-variable **si**)



Synthèse des problématiques traitées

- Faut-il des descripteurs de composants ? oui (Choix 1)
- Que mettre sur étagère : les composants ou les descripteurs ? les descripteurs (Choix 2)
- Assemble t-on les composants et/ou les descripteurs ? les composants (Choix 3)
- Faut-il des ports unidirectionnels ou bidirectionnels ? unidirectionnels (Choix 4)
- À quoi associe t-on une interface ? un port (Choix 5)
- Que décrit une interface ? un ensemble de signatures de services (Choix 6)
- Comment représenter les types de base ? par des composants (Choix 7)
- Comment uniformiser le traitement des types de bases ? introduction d'un port par défaut (Choix 8)
- Comment connecter des composants ? liaison binaire, port requis $\longrightarrow$ port fourni (Choix 9)
- Comment gérer les collections ? introduction des « multi-ports » (Choix 10)
- Comment invoquer un service ? toujours à travers un port (Choix 11)
- Comment traiter les auto-références ? introduction du port `Self` (Choix 12)
- Que peuvent-être les arguments ? des références sur des ports (Choix 13)
- Comment traiter le passage d'arguments ? par des (dé)connexions automatiques (Choix 14)
- Faut-il réifier les connexions ? oui (Choix 15)
- Qu'est-ce qu'un connecteur ? un composant (Choix 16)
- Comment faciliter l'écriture des connecteurs ? introduction d'un service par défaut (« code glue ») (Choix 17)
- Comment réutiliser des connecteurs ? standardisation des connecteurs et de leur paramétrage (Choix 18)
- Faut-il des composites ? oui (Choix 19)
- Peut-on imbriquer les descriptions de composants ? non (Choix 20)
- Comment référencer les sous-composants ? par des ports internes requis
- Comment traiter les invocations de services en tenant compte de tous les choix ? (Algorithme 1-2)

Synthèse comparative

	Julia	ArchJava	SCL
Passage d'arguments	☹	😟	☺
Uniformité	😟	☹	☺
Auto-référence	☹	☹	☺
Découplage	☺	😟	☺
Séparation des préoccupations	😟	😟	😟
Non-anticipation	😟	😟	😟

Plan

Les problèmes restants

Le problème de la séparation des préoccupations

Le code métier et le code transversal (ex : Sécurité, Journalisation) devraient être bien séparés

Le problème de la non-anticipation

Le programmeur ne devrait pas écrire de code pour permettre les connexions basées sur les changements d'état des propriétés

Les problèmes restants

Le problème de la séparation des préoccupations

Le code métier et le code transversal (ex : Sécurité, Journalisation) devraient être bien séparés

Le problème de la non-anticipation

Le programmeur ne devrait pas écrire de code pour permettre les connexions basées sur les changements d'état des propriétés

Une solution : exploiter la programmation par aspects (AOP)

- En PPO : AspectJ, HyperJ, ...
- En PPC : AspectJ2EE, FAC, FuseJ, ...

Support de l'AOP en Scl : choix et extensions

Nos choix pour SCL

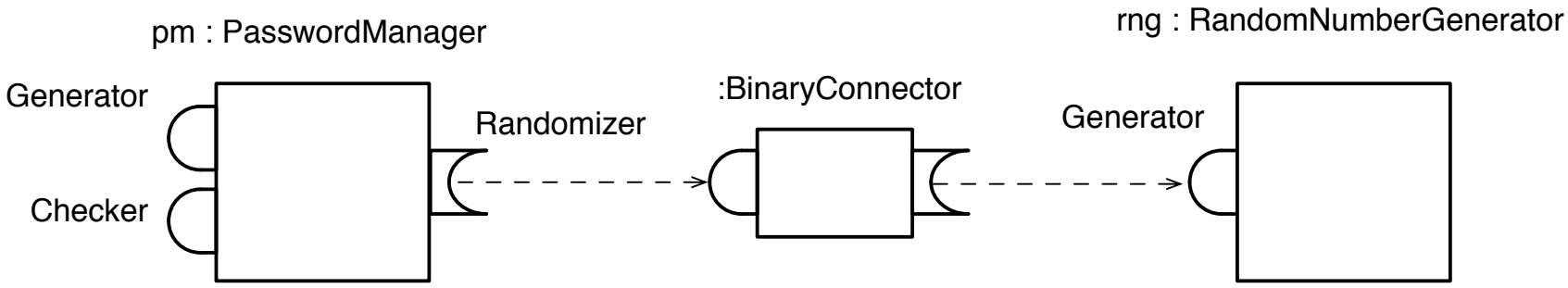
- Modèle de points de jonctions de type boîte noire (Choix 21)
- Tout composant peut être tissé (Non-anticipation) (Choix 22)
- Les services des composants sont des *advices* (Choix 23)

De nouveaux types de liaisons entre ports : les « liaisons d'aspects »

- *before/after service invocation* (`bSI`/`aSI`)
- *before/after connection* (`bCon`/`aCon`)
- *before/after disconnection* (`bDiscon`/`aDiscon`)

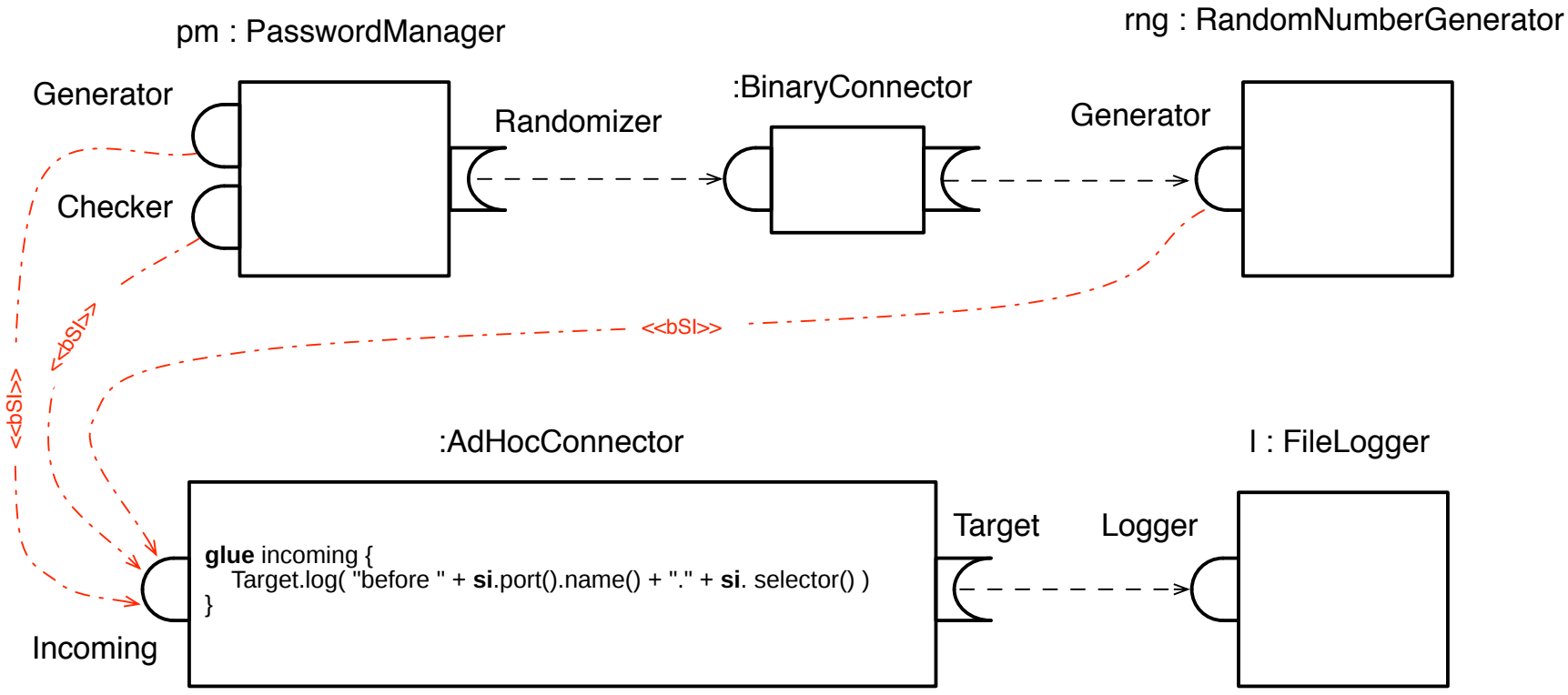
Meilleure séparation des préoccupations

Exemple de l'introduction d'une fonctionnalité de trace



Meilleure séparation des préoccupations

Exemple de l'introduction d'une fonctionnalité de trace



Respect de la non-anticipation

grâce aux « propriétés observables » et aux liaisons d'aspects

1. Code écrit par le programmeur

```
interface IChat {
join(serverAddress);
leave; send(message) }

descriptor ChatClient {
   providedport Chat, IChat

   property ChatText {
      accessport AccessCT, { getChatText; setChattext(v) }
   }
   ...
}
```

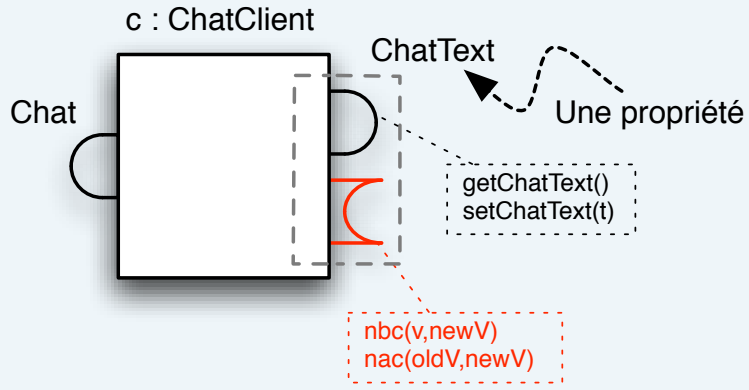
Respect de la non-anticipation

grâce aux « propriétés observables » et aux liaisons d'aspects

1. Code écrit par le programmeur

```
interface IChat {
join(serverAddress);
leave; send(message) }

descriptor ChatClient {
   providedport Chat, IChat

   property ChatText {
      accessport AccessCT, { getChatText; setChattext(v) }
   }
   ...
}
```

2. Composant mis sur étagère



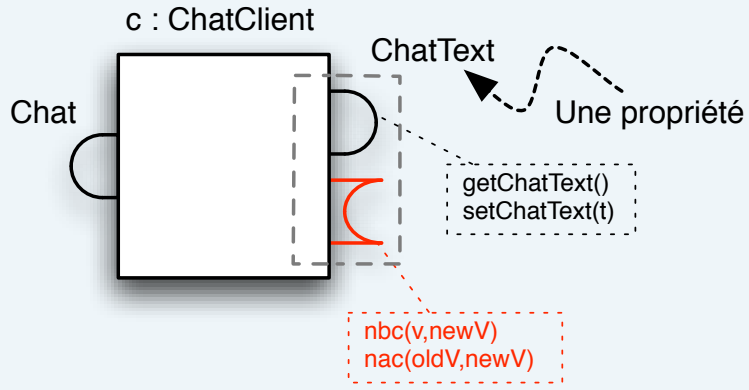
Respect de la non-anticipation

grâce aux « propriétés observables » et aux liaisons d'aspects

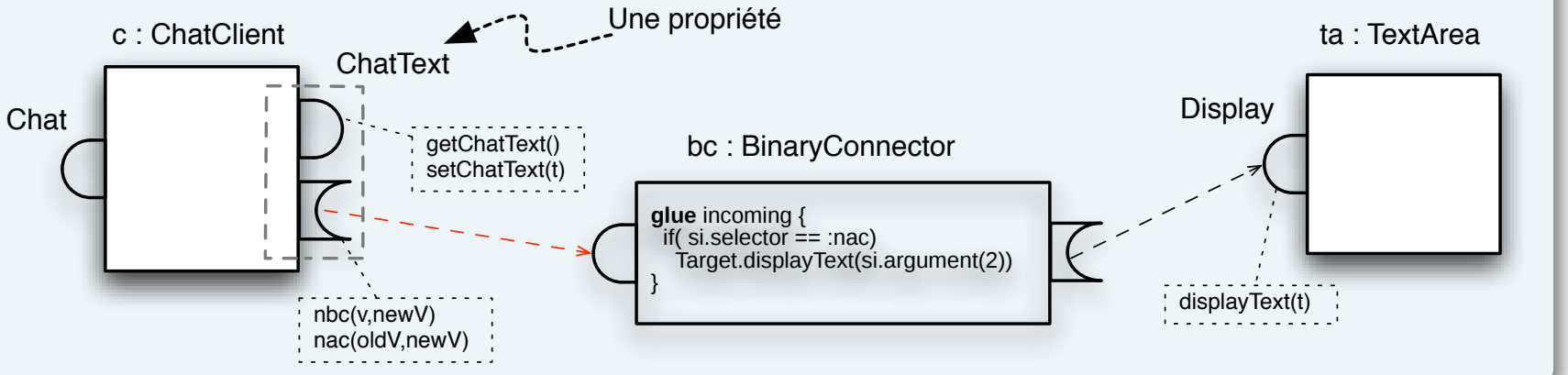
1. Code écrit par le programmeur

```
interface IChat {
join(serverAddress);
leave; send(message) }

descriptor ChatClient {
   providedport Chat, IChat

   property ChatText {
      accessport AccessCT, { getChatText; setChattext(v) }
   }
   ...
}
```

2. Composant mis sur étagère

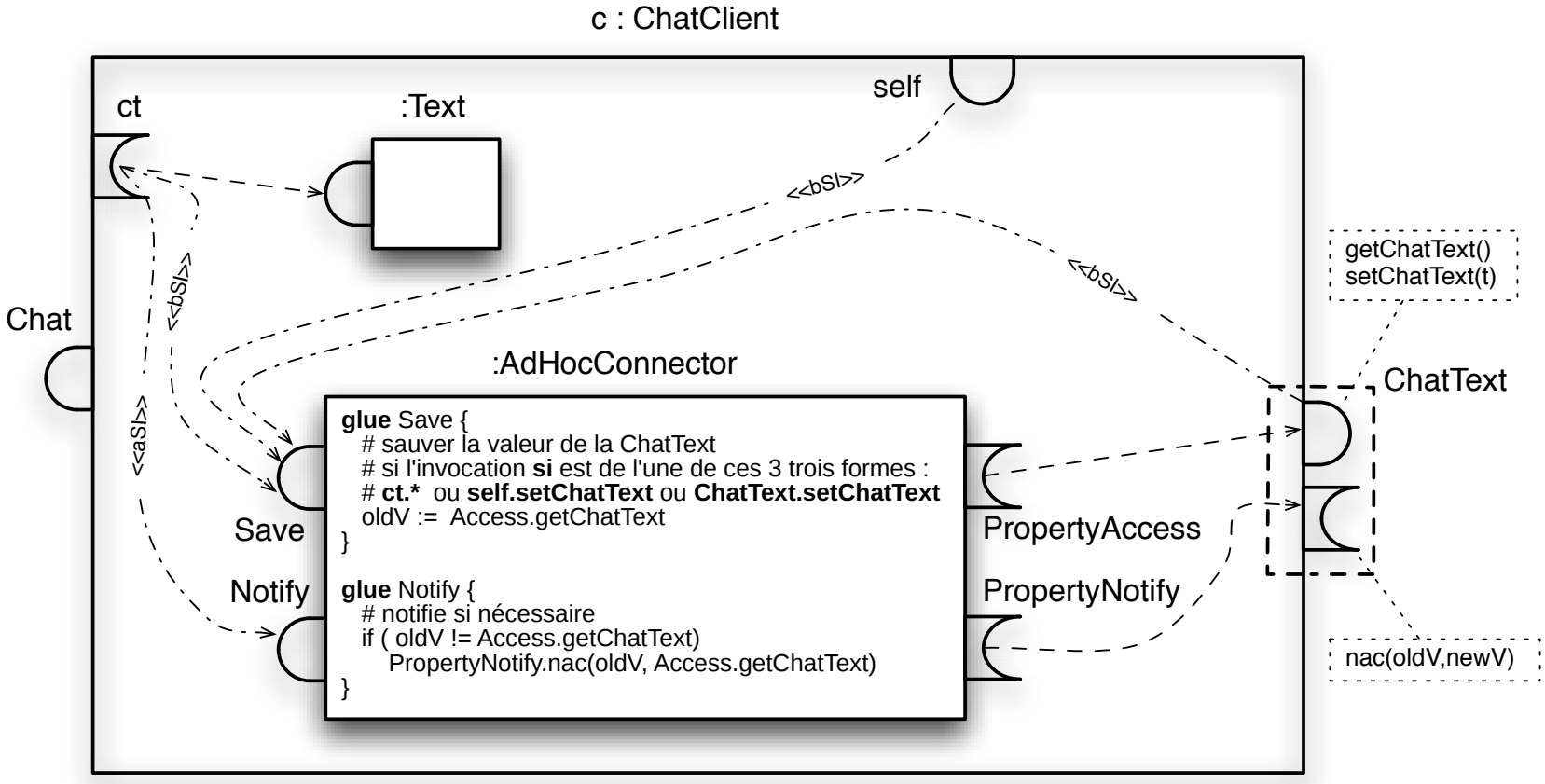


3. Connexion standard par l'architecte



Mise en place automatique du port de notification

grâce aux liaisons d'aspects



Synthèse comparative

Bilan

- Possibilité de factoriser le code transversal
- Possibilité d'établir des connexions basées sur les changements d'états des propriétés observables sans qu'aucun code ne soit nécessaire

Synthèse comparative

Bilan

- Possibilité de factoriser le code transversal
- Possibilité d'établir des connexions basées sur les changements d'états des propriétés observables sans qu'aucun code ne soit nécessaire

	Julia	ArchJava	SCL
Passage d'arguments	☹	😟	☺
Uniformité	😟	☹	☺
Auto-référence	☹	☹	☺
Découplage	☺	😟	☺
Séparation des préoccupations	😟	😟	☺
Non-anticipation	😟	😟	☺

Plan

Deux Prototypes : en Smalltalk et en Ruby

Pourquoi en Smalltalk ?

- Pas Java
- Typage dynamique
- Un MOP puissant
- Une syntaxe unique

Pourquoi Ruby et pourquoi un deuxième prototype ?

- Pas Java
- Typage dynamique
- Un MOP puissant
- Une syntaxe plus facilement compréhensible

Choix d'implémentation

Techniques utilisées pour implanter les *Domain Specific Languages*

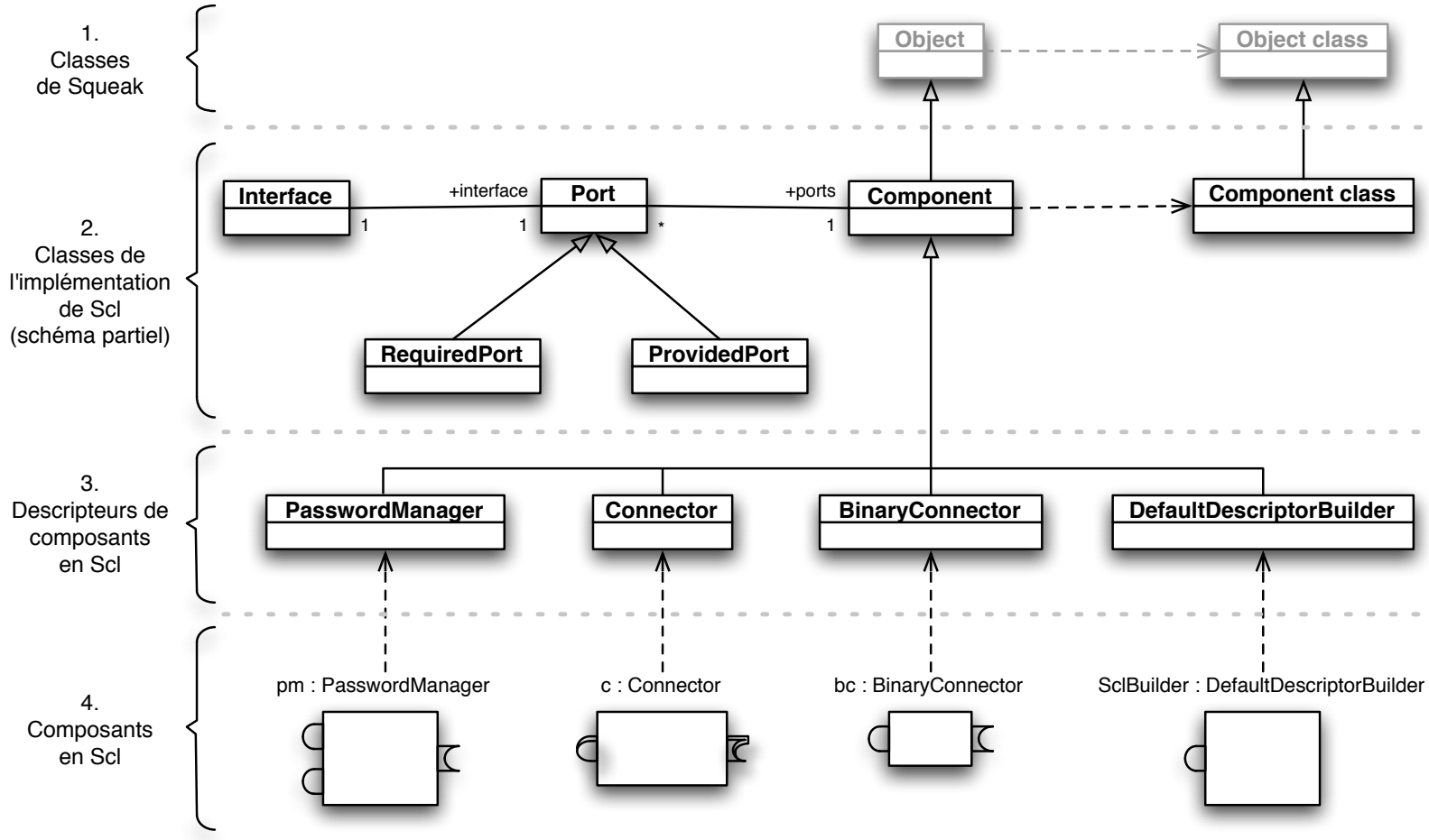
L'idée

- Pas d'analyseur syntaxique, lexical
- Utilisation des constructions syntaxiques du langage hôte
- Exploitation de la puissance du MOP et des capacités du langage hôte (ex : fermetures lexicales)

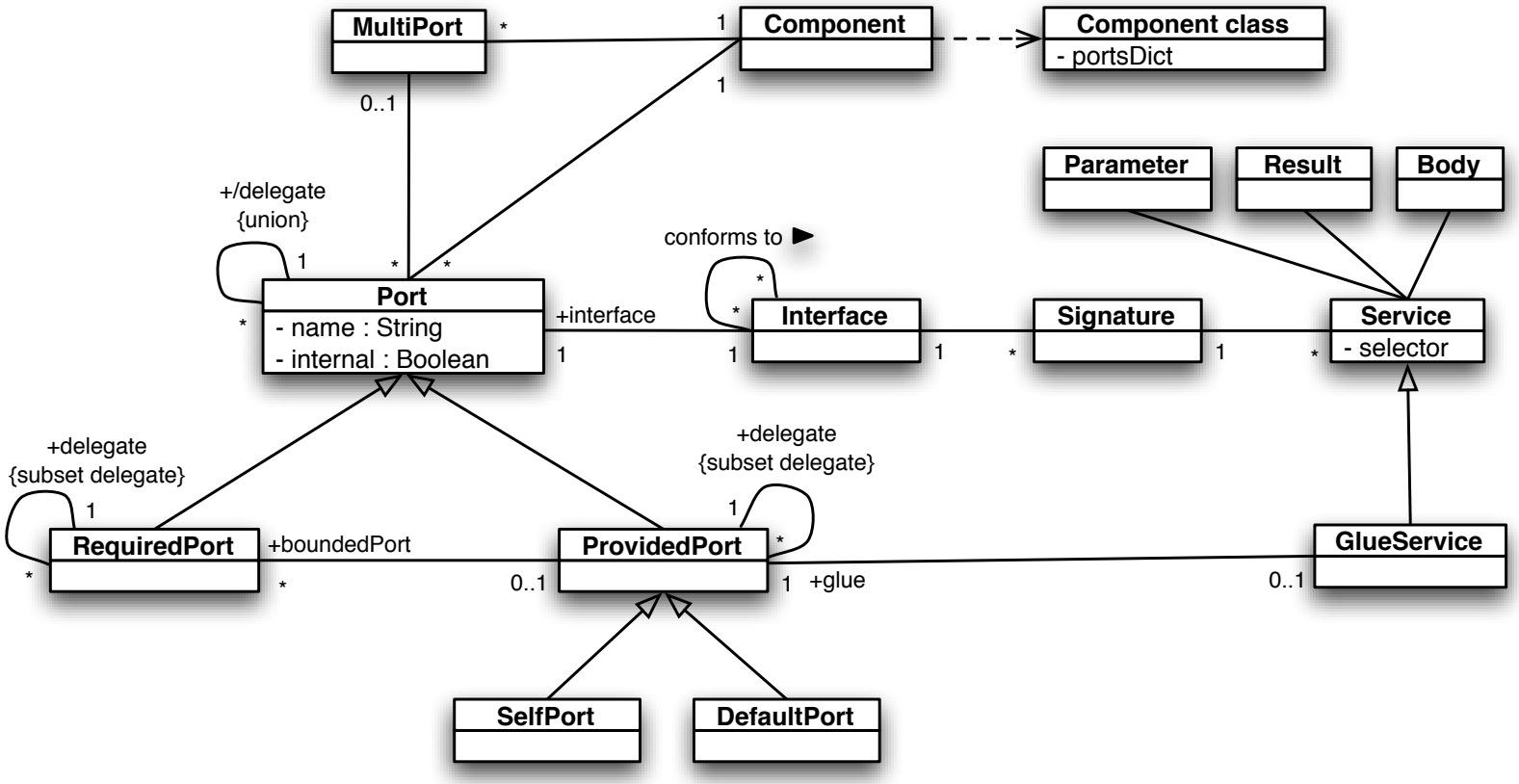
Avantages/Inconvénients

- ☺ Prototypage rapide
- ☺ Évolutions faciles
- ☺ Réutilisation des outils du langage hôte (ex : *debugger*)
- ☹ Syntaxe obligatoirement compatible avec celle du langage hôte
- ☹ Langage développé dépendant du langage hôte
- ☹ Efficacité

Architecture générale du prototype en Smalltalk



Niveau 2 : Diagramme UML de l'implantation



Au cœur de la syntaxe

```
(SclBuilder newDescriptor: #PasswordManager)
    requiredPort: #Randomizer -> {#getRandomNumber};
    providedPort: #Generator -> {#generatePwd:. #generateADigitsOnlyPwd:};
    providedPort: #Checker -> {#isValid:}.

PasswordManager>>generatePwd: size
    "..."
    i:= Randomizer getRandomNumber
    "..."

"connexion"
pm := PasswordManager newC.
rng := RandomNumberGenerator newC.

(pm port: #Randomizer) bindTo: (rng port: #Generator)
glue: [ :source :target :si |
    ^target random * 26
]
```

Quelques détails d'implantation

- Interception des envois de messages
- Utilisation des blocks
- Primitives **newC**, **bindTo:**, **bindTo:glue:**

Détails sur l'implantation de l'invocation de service

```
Port>>doesnotunderstand: message
^self invoke: (ServiceInvocation newFrom: message sentThrough: self)

Port>>invoke: aServiceInvocation
   | res receiverObject |
   (self interface selectors includes: aServiceInvocation selector) ifFalse: [
    ^super doesNotUnderstand: aServiceInvocation asMessage
   ].
   receiverObject := self computeReceiverObject
   self processBeforeServiceInvocationBindings: aServiceInvocation.
   res :=  aServiceInvocation sendTo: receiverObject.
   (res isKindOf: Port) ifFalse: [ res := self ].
   aServiceInvocation result: res.
   self processAfterServiceInvocationBindings: aServiceInvocation.
   ^res

Port>>computeReceiverObject
   self subclassResponsability

RequiredPort>>computeReceiverObject
   (self isBound)
      ifTrue:  [ ^self boundPort ]
      ifFalse: [
         (self isDelegated)
         ifTrue:[ ^self delegate ]
         ifFalse: [ self error: self name, ' : not bound and not delegated' ]
      ]

ProvidedPort>>computeReceiverObject
   (self isDelegated)
      ifTrue:[ ^self delegate]
      ifFalse: [ ^self component]
```

Bilan

L'avancement des prototypes

	Smalltalk	Ruby
Bootstrap	✓	✓
Unification objets/composants	✓	✗
Liaisons de ports	✓	✓
« Code glue »	✓	✗
Liaisons d'aspects	✓	✗
Propriétés	✓	✗

Plan

Conclusion

Analyse comparative de l'existant

- Diversité des concepts
- Mise en pratique difficile de la PPC
- Sous-spécifications et non-uniformité des LACs

Le langage à composants SCL

- ✓ Respecte le découplage et la non-anticipation
- ✓ Permet la séparation des préoccupations
- ✓ Est minimal tout en étant un langage utilisable
- ✓ Est uniforme
- ✓ Est opérationnel

Publications

Luc Fabresse, Christophe Dony, et Marianne Huchard.

Foundations of a Simple and Unified Component-Oriented Language.
Journal of Computer Languages, Systems & Structures, 2007. doi :10.1016/j.cl.2007.05.002.

Luc Fabresse, Christophe Dony, et Marianne Huchard.

SCL : a Simple, Uniform and Operational Language for Component-Oriented Programming in Smalltalk.
In *Advances in Smaltalk, Proceedings of 14th International Smalltalk Conference (ISC), September 4–8, 2006*, éditeur De Meuter, Wolfgang, volume 4406, pages 91–110. LNCS, Springer-Verlag, April 2007.

Luc Fabresse, Christophe Dony, et Marianne Huchard.

Unanticipated Connection of Components based on their State Changes Notifications.
In *International Workshop on Evaluation and Evolution of Component Composition (EECC'06). In proceedings of 18th International Conference on Software Engineering and Knowledge Engineering (SEKE 2006)*, pages 702–708. Knowledge System Institute (KSI), 5–7 July 2006.

Luc Fabresse, Christophe Dony, et Marianne Huchard.

Connexion non-anticipée de composants en SCL : Une voie pour l'évolution des logiciels.
In *Atelier sur l'Evolution du Logiciel*, pages 1–7, 21 Mars 2006.

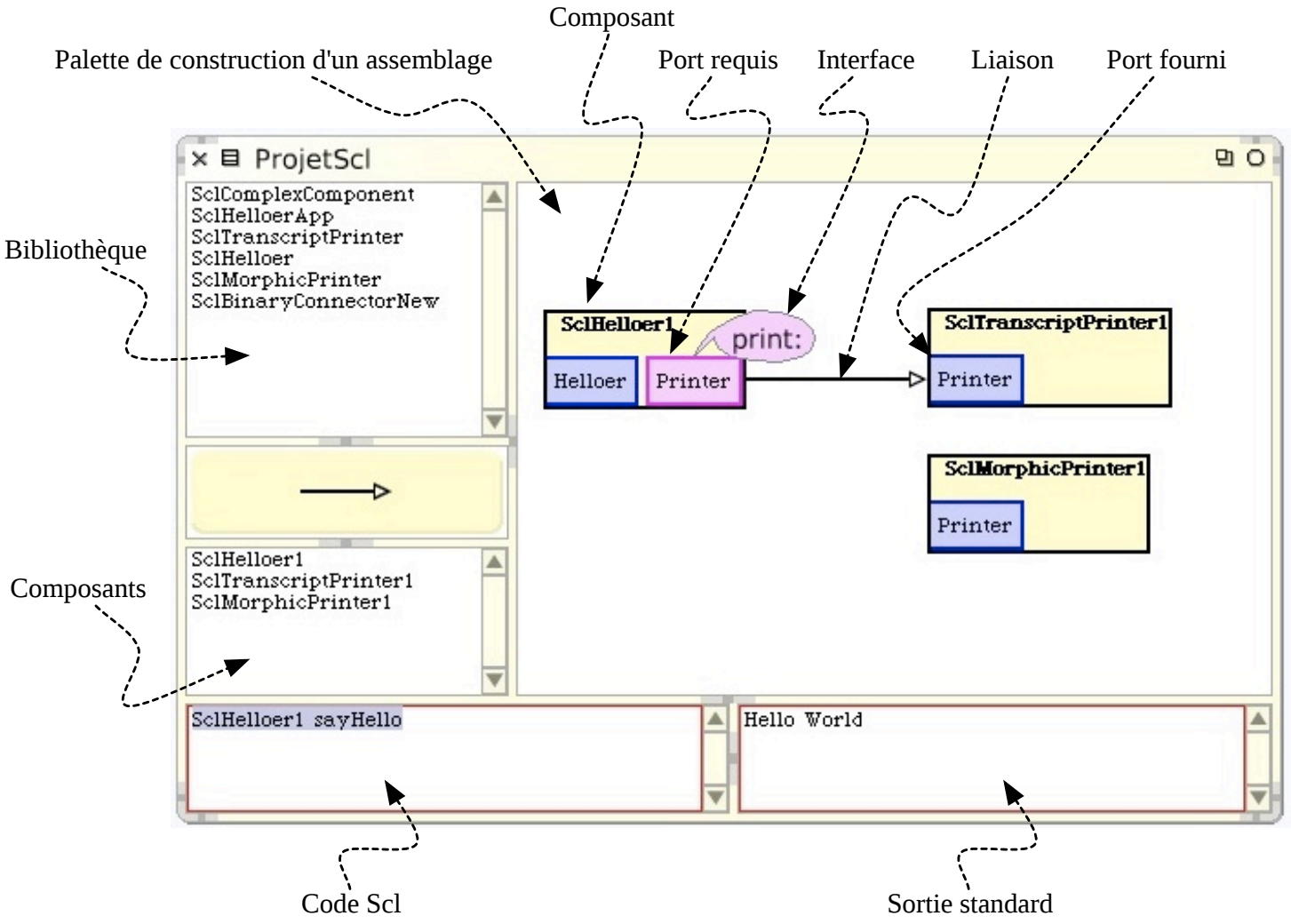
Luc Fabresse.

Programmation de composants interfaçables.
In *Actes de JOCM, Journées du groupe Objets, Composants et Modèles*, pages 19–24, 16 Mars 2004.

Luc Fabresse, Christophe Dony, Marianne Huchard, et Olivier Pichon.

Vers des composants logiciels interfaçables.
In *8ème Colloque Agents Logiciels, Coopération, Apprentissage et Activité humaine (ALCAA'04)*, pages 33–48, 17–18 Juin 2004.

Perspective : Vers un environnement de développement graphique



Perspectives

Perspectives centrées sur SCL

- Formalisation : expliciter la sémantique opérationnelle
- Améliorer la vérification des assemblages
- Proposer une version réflexive de SCL

Perspectives autour de SCL

- Relations avec les services Web
- Transformations de modèles

Fin...

Une idée de slogan pour SCL ?

Write Once, Assembly Everywhere ☺

Merci pour votre attention !

```
Person allInstances do: [ :p |
 luc thanks: p for: #(#Coming #Listening).
 luc congrates: p because: #HasStayedUntilHere
]
```