

Génie Logiciel

Guillaume Baudart

Bibliographie

Software Engineering, 10th Edition
I. Sommerville, 2016

Software Engineering, A practitioner's Approach, 8th Edition
R. Pressman, B. Maxim, 2015

Design Patterns, Element of Reusable Object-Oriented Software
E. Gamma, R. Helm, R. Johnson, J. Vlissides, 1994

Cours de Génie Logiciel Avancé, Université Paris Diderot
S. Zacchiroli, Mihaela Sighireanu 2014

Introduction

Génie logiciel

Définitions

software. Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system. *See also:* **application software; support software; system software.** *Contrast with:* **hardware.**

Définitions

software. Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system. *See also: application software; support software; system software. Contrast with: hardware.*

software engineering. (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
(2) The study of approaches as in (1).

Définitions

software life cycle. The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software life cycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and, sometimes, retirement phase. *Note:* These phases may overlap or be performed iteratively. *Contrast with:* **software development cycle.**

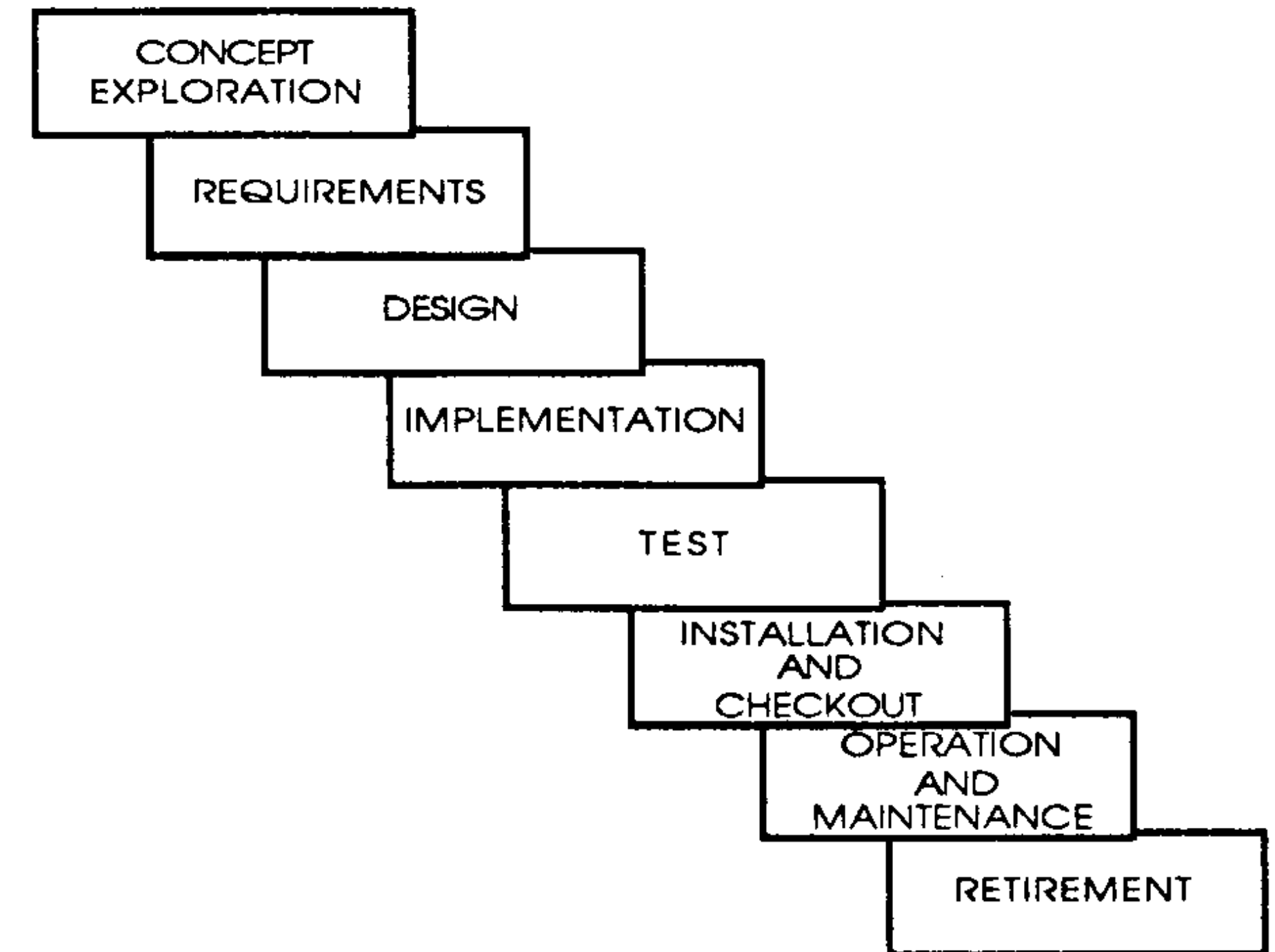


Fig 15
Sample Software Life Cycle

Définitions

software development cycle. The period of time that begins with the decision to develop a software product and ends when the software is delivered. This cycle typically includes a requirements phase, design phase, implementation phase, test phase, and sometimes, installation and checkout phase.

Contrast with: **software life cycle.**

Notes: (1) The phases listed above may overlap or be performed iteratively, depending upon the software development approach used.

(2) This term is sometimes used to mean a longer period of time, either the period that ends when the software is no longer being enhanced by the developer, or the entire software life cycle.

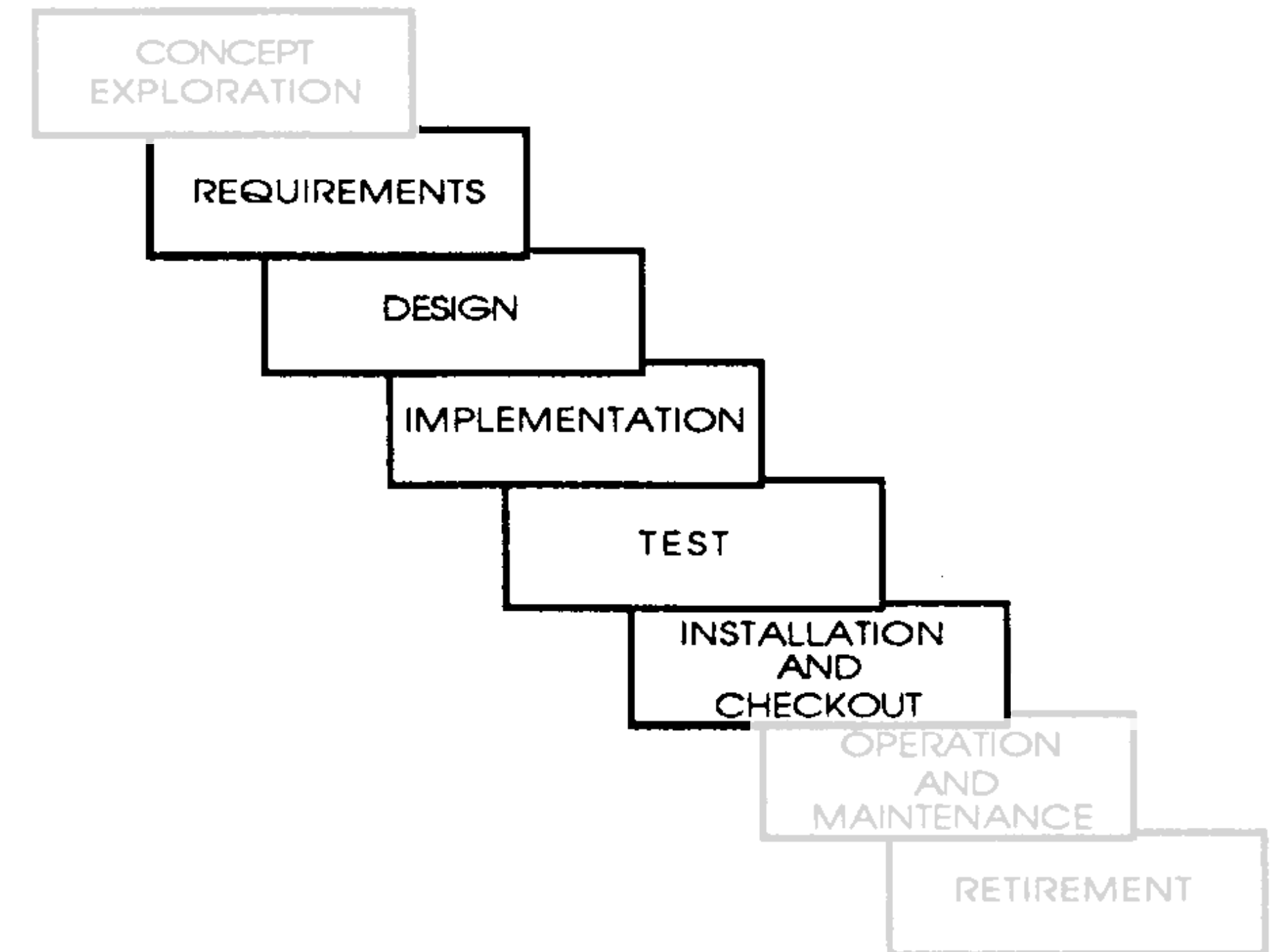


Fig 15
Sample Software ~~Life Cycle~~
development cycle

Problèmes ?

Problèmes ?

Logiciels de plus en plus présents

- Communication (téléphone, internet)
- Infrastructure (électricité, gaz, hôpitaux)
- Transport (voiture, train, avion)

Problèmes ?

Logiciels de plus en plus présents

- Communication (téléphone, internet)
- Infrastructure (électricité, gaz, hôpitaux)
- Transport (voiture, train, avion)

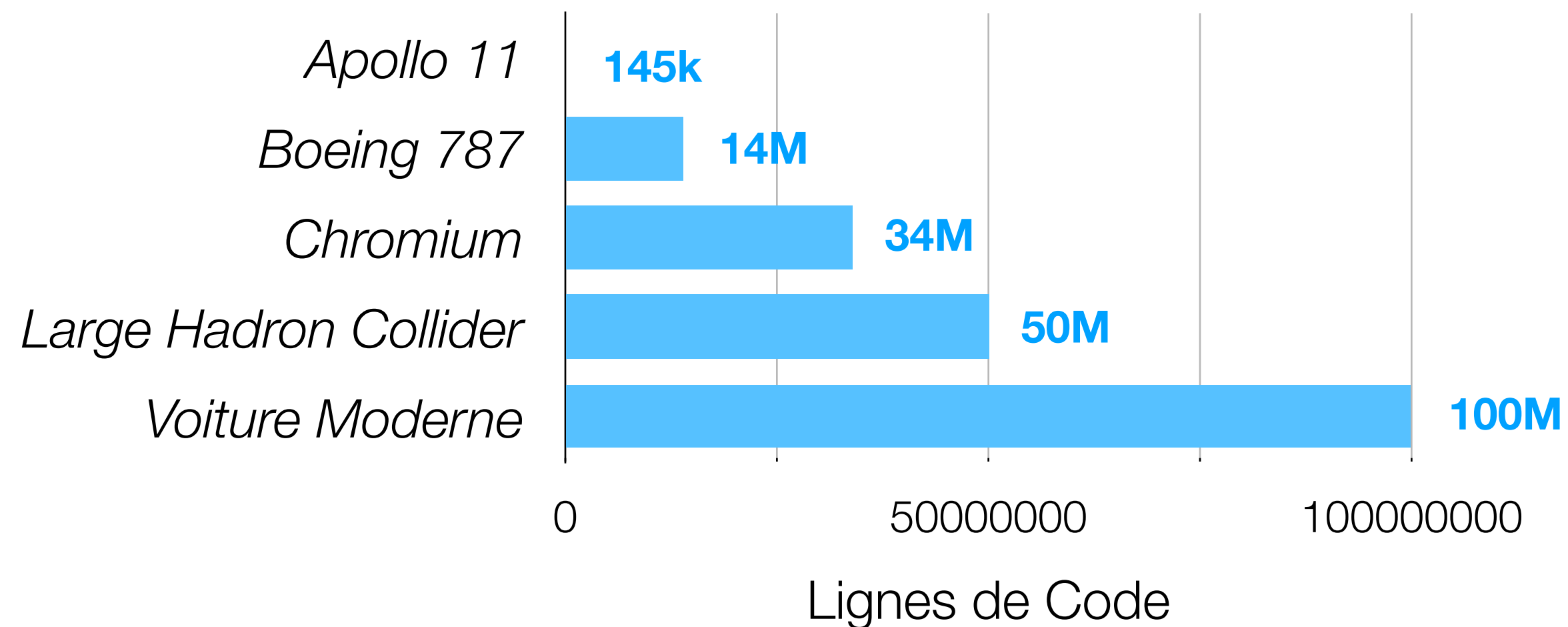
Taille et complexité en constante augmentation

Problèmes ?

Logiciels de plus en plus présents

- Communication (téléphone, internet)
- Infrastructure (électricité, gaz, hôpitaux)
- Transport (voiture, train, avion)

Taille et complexité en constante augmentation

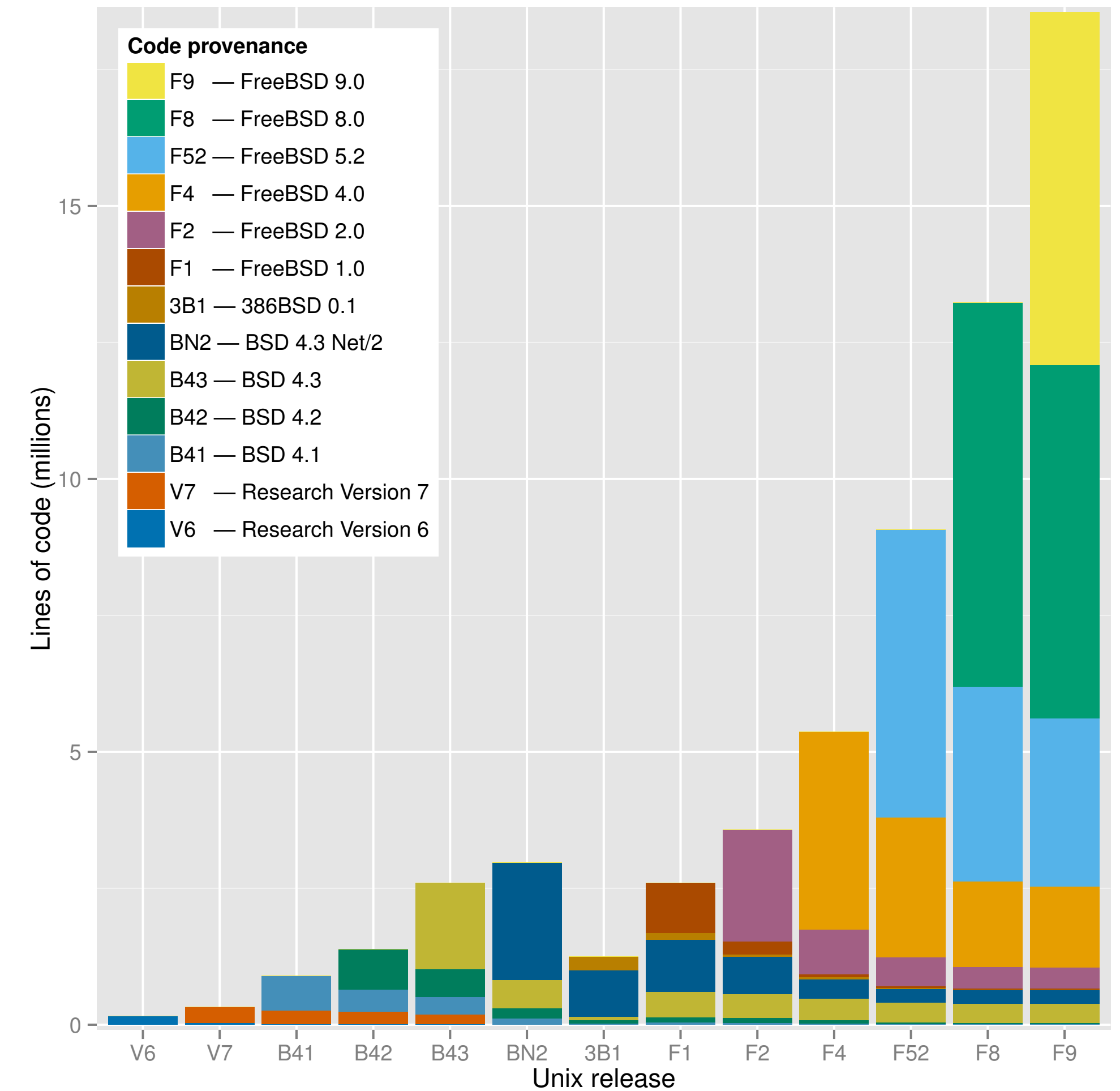
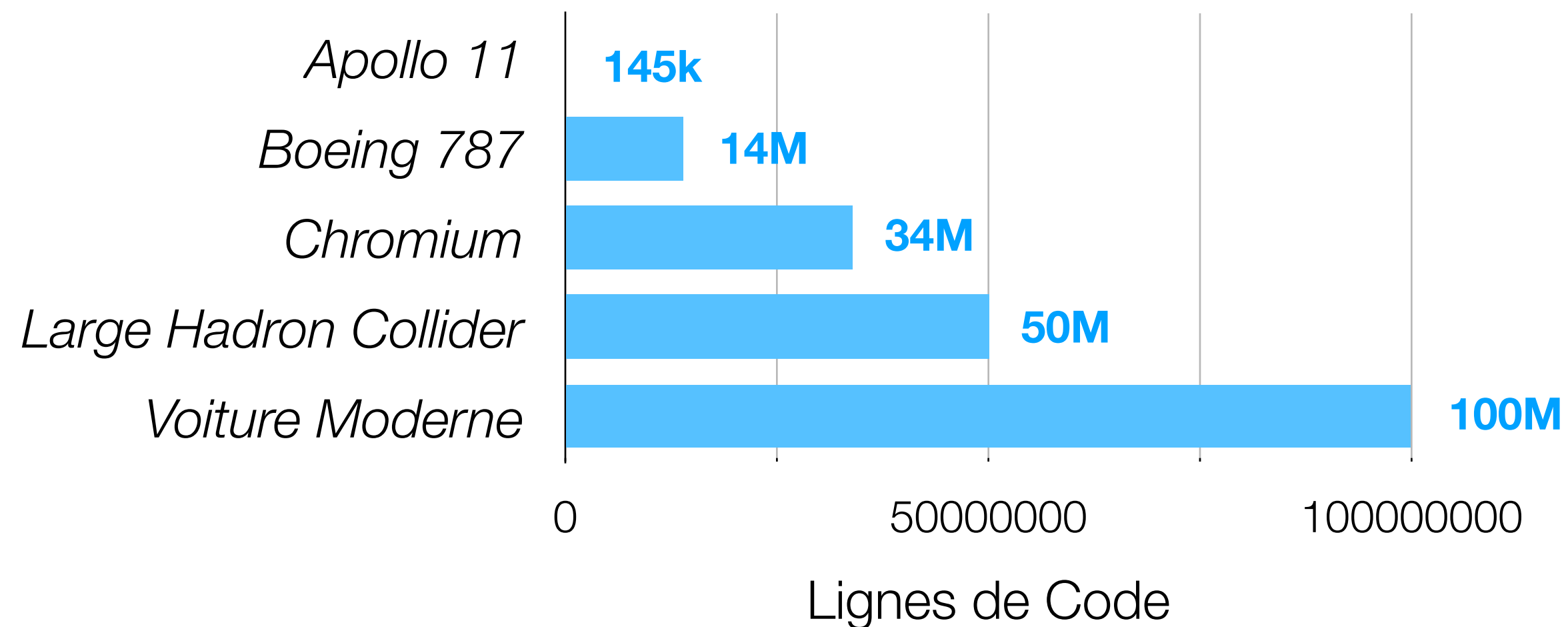


Problèmes ?

Logiciels de plus en plus présents

- Communication (téléphone, internet)
- Infrastructure (électricité, gaz, hôpitaux)
- Transport (voiture, train, avion)

Taille et complexité en constante augmentation

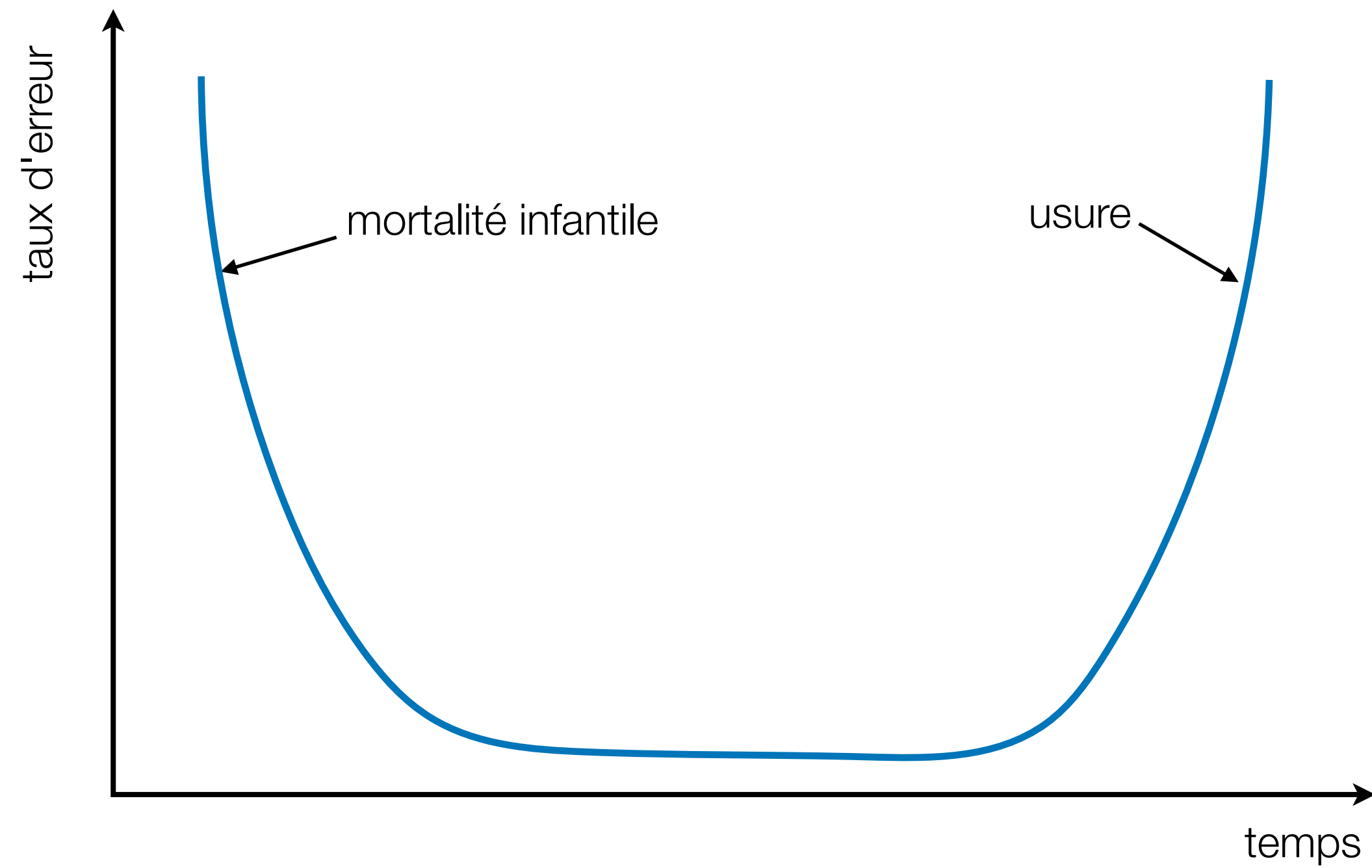


1972
5 kLoC

2015
26 MLoC

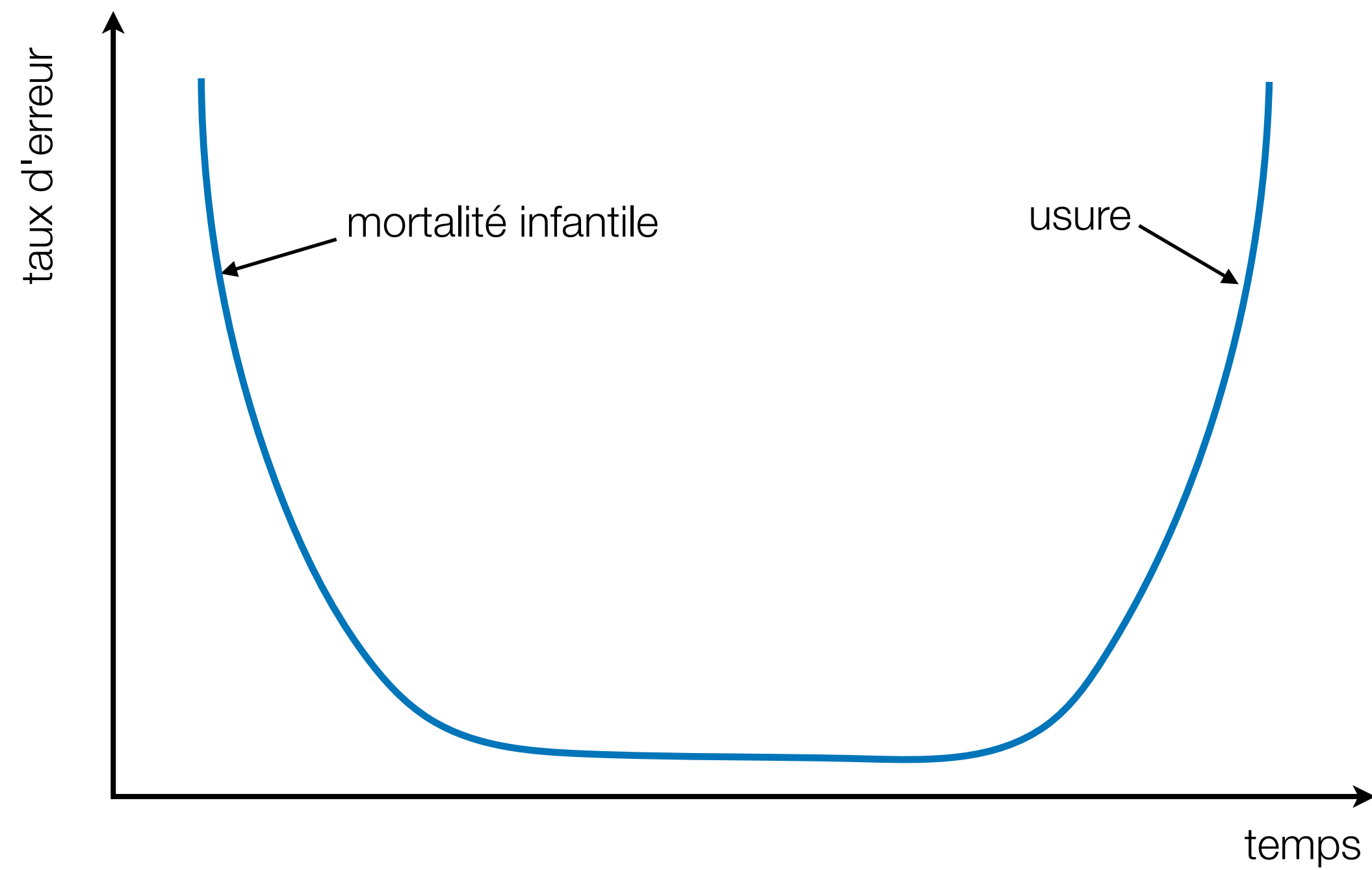
Problèmes ?

Évolution matérielle



Problèmes ?

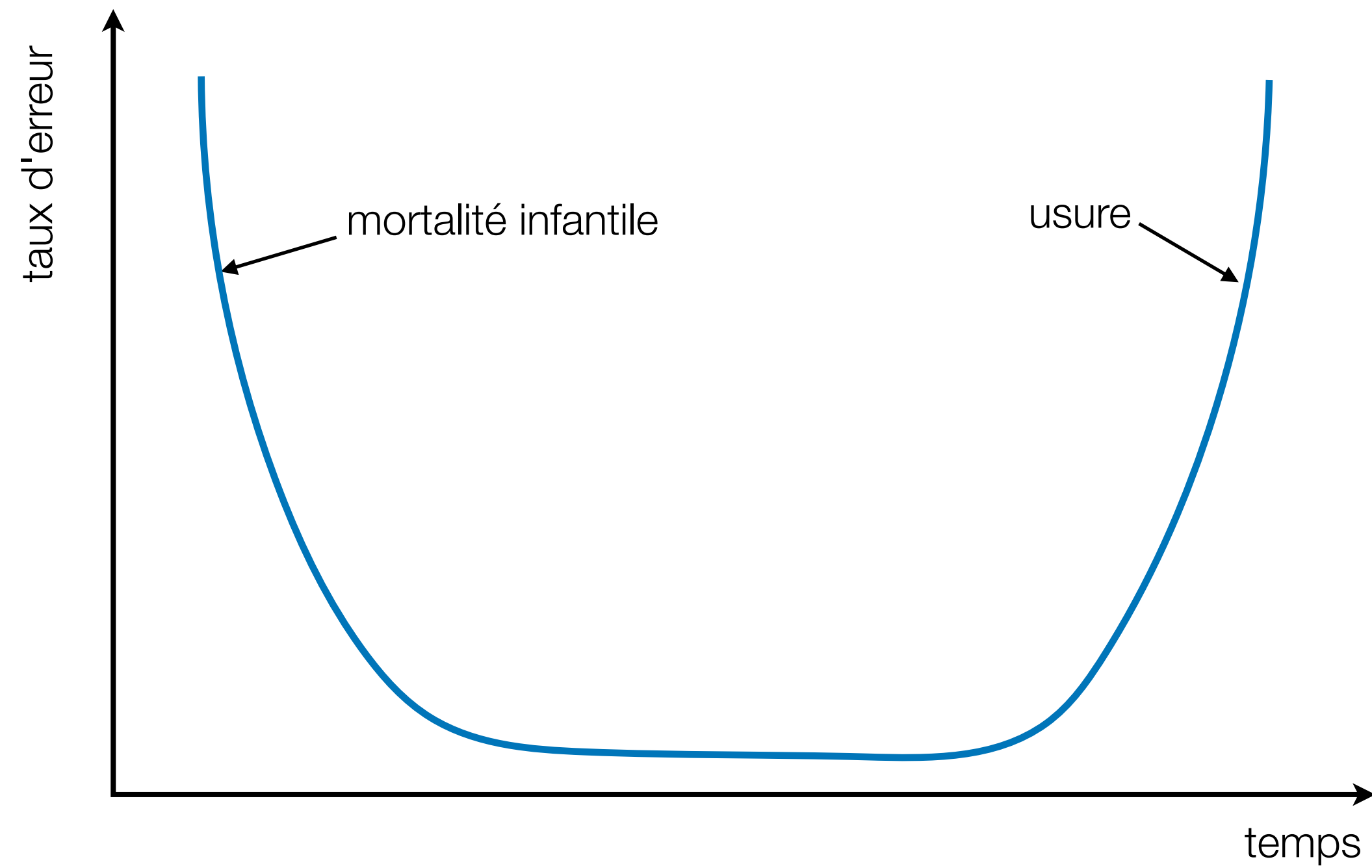
Évolution matérielle



Un logiciel ne s'use pas !

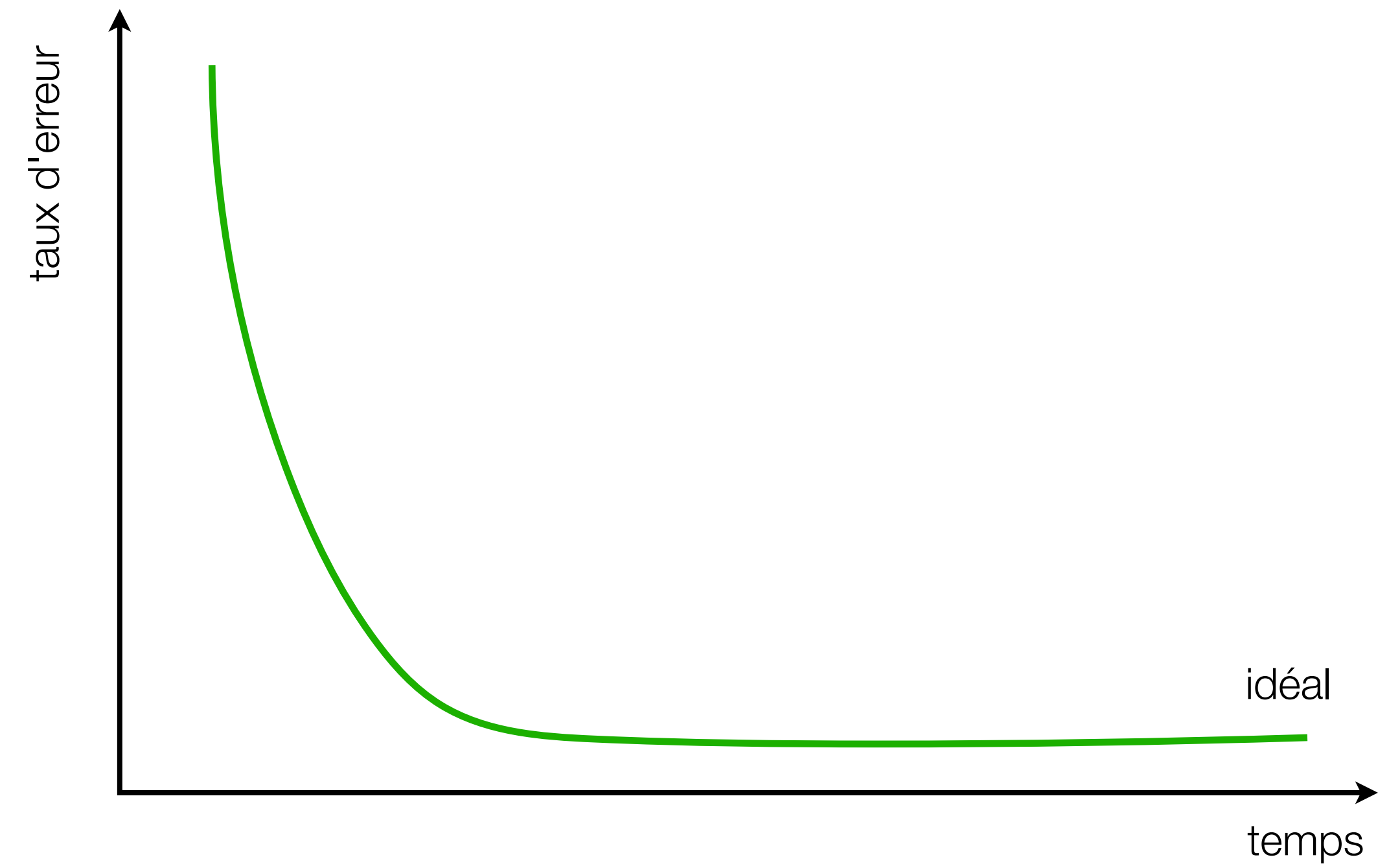
Problèmes ?

Évolution matérielle



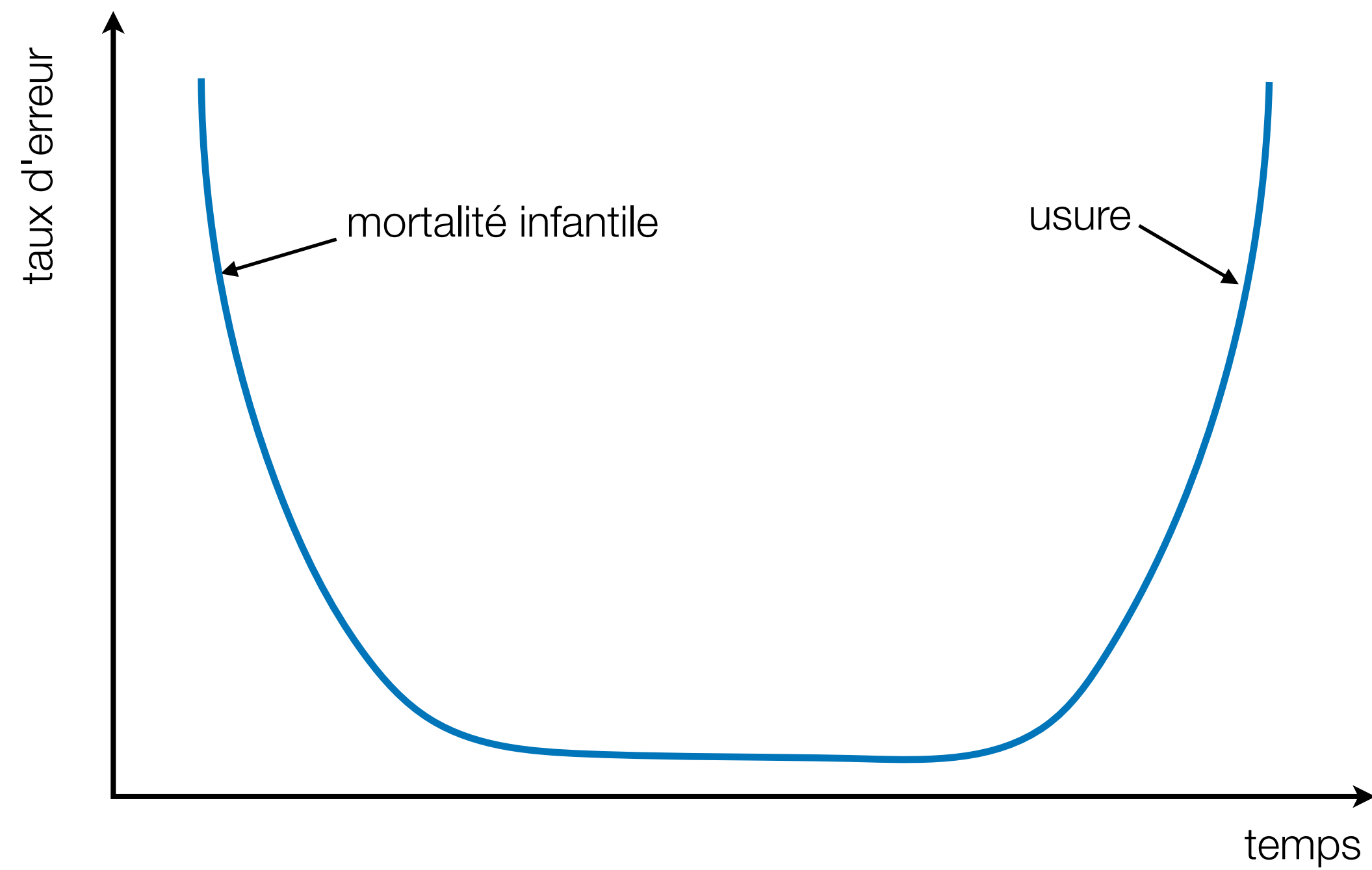
Un logiciel ne s'use pas !

Évolution Logiciel



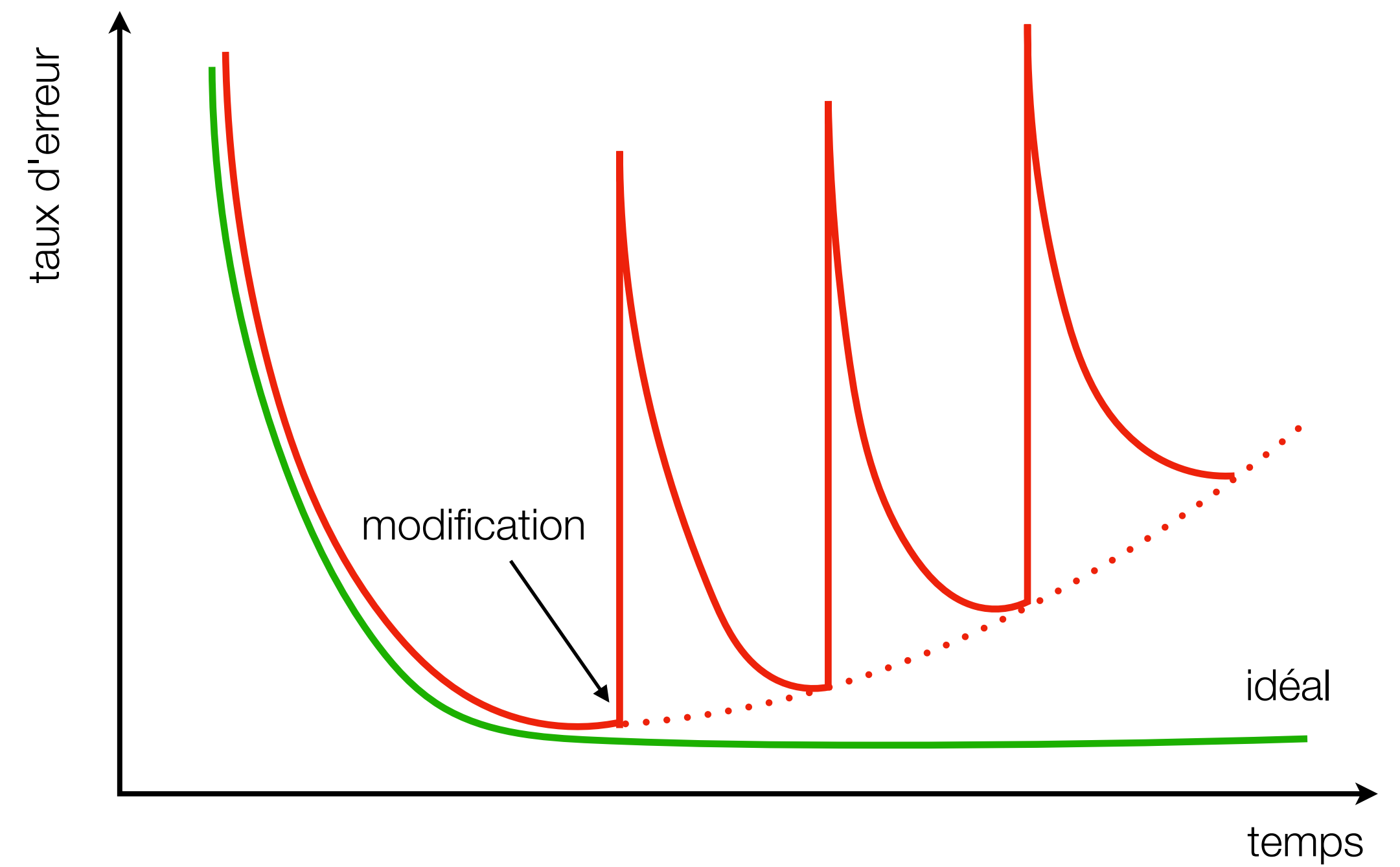
Problèmes ?

Évolution matérielle



Un logiciel ne s'use pas !

Évolution Logiciel



La plupart des logiciels évoluent...

Génie logiciel

Génie logiciel

Objectifs : Aider la réalisation de logiciels de grande taille

- Spécifications conforme à ce qui est attendu
- Implémentation conforme à la spécification
- Délais et coût de production maîtrisés
- Faciliter la maintenance et les évolutions

Génie logiciel

Objectifs : Aider la réalisation de logiciels de grande taille

- Spécifications conforme à ce qui est attendu
- Implémentation conforme à la spécification
- Délais et coût de production maîtrisés
- Faciliter la maintenance et les évolutions

En pratique

- Principes : pratique guidé par le bon sens
- Méthodes : cycle de développement, gestion de projet
- Techniques : pair programming, TDD, méthodes formelles, ...
- Outils : Git, UML, outils de test, assistants de preuve, ...

Grands principes

Grands principes

Ghezzi et al., *Fundamentals of Software Engineering*, 1991

Grands principes

Ghezzi et al., *Fundamentals of Software Engineering*, 1991

- Rigueur

Grands principes

Ghezzi et al., *Fundamentals of Software Engineering*, 1991

- Rigueur
- Décomposition en sous problèmes indépendants

Grands principes

Ghezzi et al., *Fundamentals of Software Engineering*, 1991

- Rigueur
- Décomposition en sous problèmes indépendants
- Modularité

Grands principes

Ghezzi et al., *Fundamentals of Software Engineering*, 1991

- Rigueur
- Décomposition en sous problèmes indépendants
- Modularité
- Abstraction

Grands principes

Ghezzi et al., *Fundamentals of Software Engineering*, 1991

- Rigueur
- Décomposition en sous problèmes indépendants
- Modularité
- Abstraction
- Anticipation des évolutions

Grands principes

Ghezzi et al., *Fundamentals of Software Engineering*, 1991

- Rigueur
- Décomposition en sous problèmes indépendants
- Modularité
- Abstraction
- Anticipation des évolutions
- Généricité

Grands principes

Ghezzi et al., *Fundamentals of Software Engineering*, 1991

- Rigueur
- Décomposition en sous problèmes indépendants
- Modularité
- Abstraction
- Anticipation des évolutions
- Généricité
- Construction incrémentale

Grands principes

Ghezzi et al., *Fundamentals of Software Engineering*, 1991

- Rigueur
- Décomposition en sous problèmes indépendants
- Modularité
- Abstraction
- Anticipation des évolutions
- Généricité
- Construction incrémentale

Grands principes

Ghezzi et al., *Fundamentals of Software Engineering*, 1991

- Rigueur
- Décomposition en sous problèmes indépendants
- Modularité
- Abstraction
- Anticipation des évolutions
- Généricité
- Construction incrémentale

KISS
Keep It Simple, Stupid

DRY
Don't Repeat Yourself

YAGNI
You Ain't Gonna Need It

SOC
Separation Of Concerns

Processus de développement

Génie logiciel

Processus de développement

Processus de développement

Pas de solution unique

- Dépend du domaine d'application (avion ou web app)
- Dépend des attentes (certification système critique ou application android)
- Dépend aussi du cadre : budget, équipe, ...

Processus de développement

Pas de solution unique

- Dépend du domaine d'application (avion ou web app)
- Dépend des attentes (certification système critique ou application android)
- Dépend aussi du cadre : budget, équipe, ...

Activités

Exigences

Architecture

Implémentation

Validation

Maintenance

Processus de développement

Pas de solution unique

- Dépend du domaine d'application (avion ou web app)
- Dépend des attentes (certification système critique ou application android)
- Dépend aussi du cadre : budget, équipe, ...

Activités

Exigences

Architecture

Implémentation

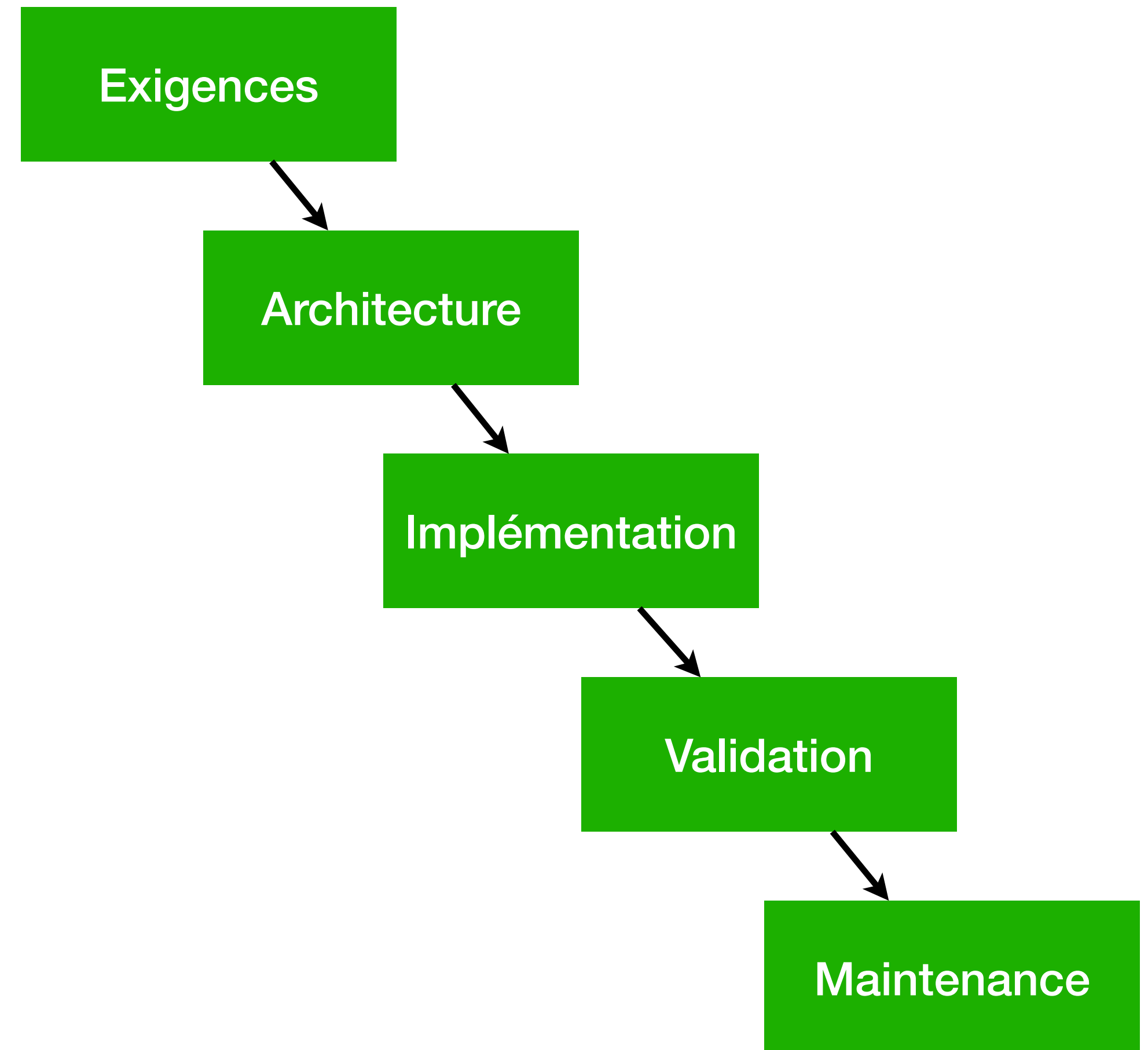
Validation

Maintenance

Processus : organiser ces activités.

- Cycle en cascade, cycle en V
- Cycle itératif
- Modèle par composants

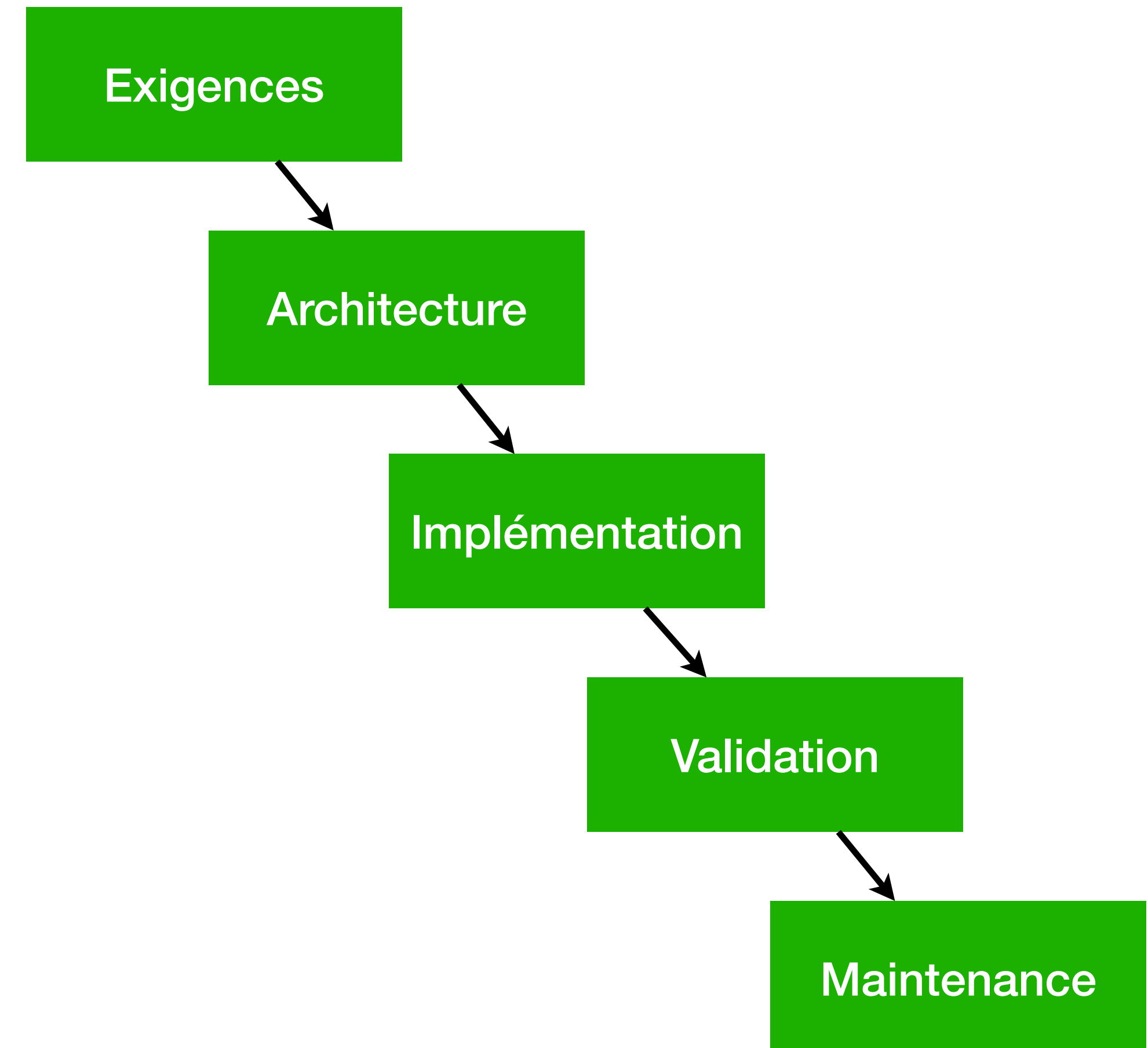
Cycle en Cascade



Cycle en Cascade

Modèle Linéaire

- Un des premiers modèle publié (Royce 1970)
- Hérité de méthodes d'ingénierie classiques
- Terminer une phase pour commencer la suivante
- Documentation à tous les niveaux



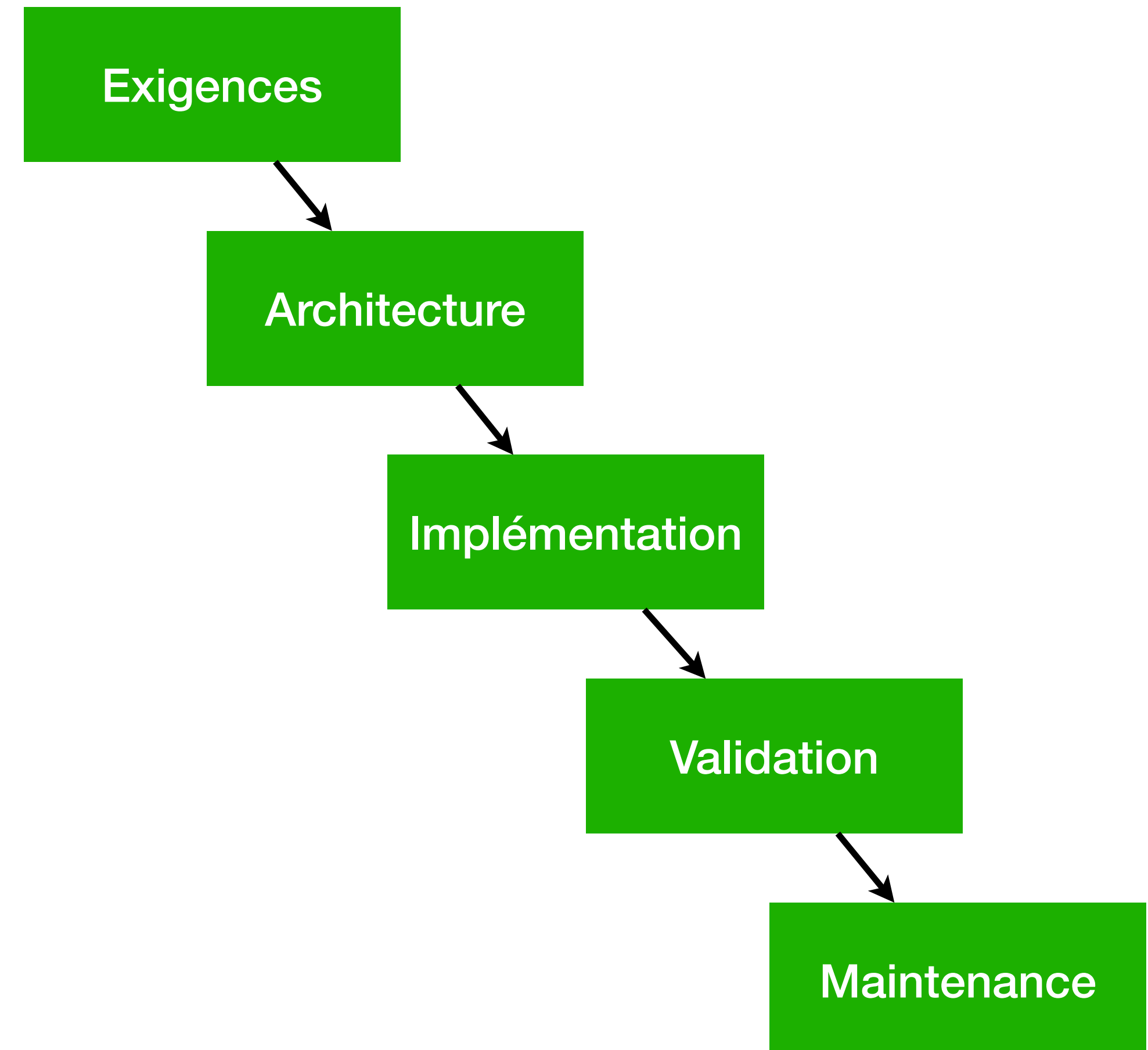
Cycle en Cascade

Modèle Linéaire

- Un des premiers modèle publié (Royce 1970)
- Hérité de méthodes d'ingénierie classiques
- Terminer une phase pour commencer la suivante
- Documentation à tous les niveaux

Avantages

- Adapté pour les projets bien définis dès le début
- Systèmes embarqués (contrôleur hardware)
- Systèmes critiques (avion, train, etc...)



Cycle en Cascade

Modèle Linéaire

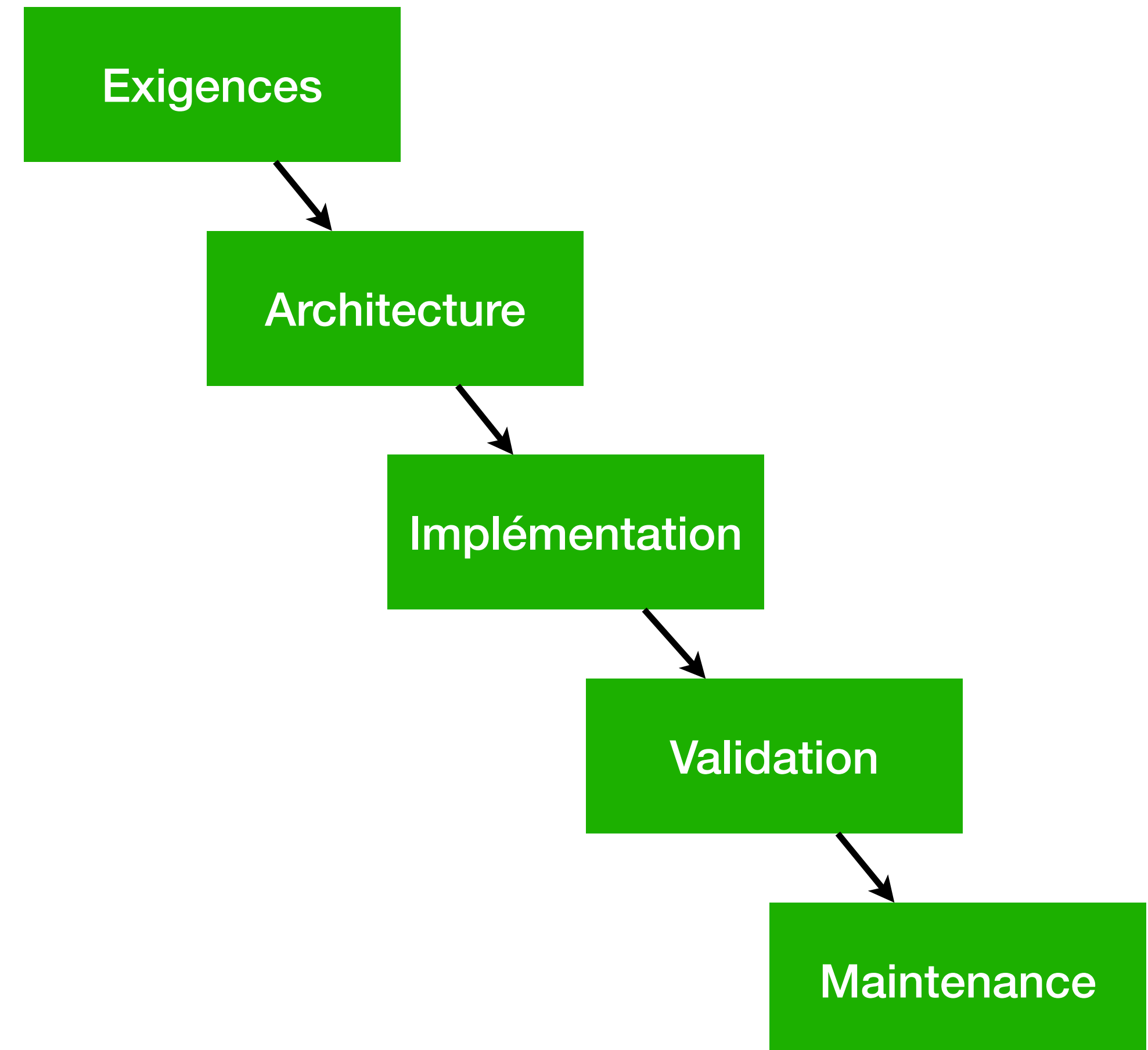
- Un des premiers modèle publié (Royce 1970)
- Hérité de méthodes d'ingénierie classiques
- Terminer une phase pour commencer la suivante
- Documentation à tous les niveaux

Avantages

- Adapté pour les projets bien définis dès le début
- Systèmes embarqués (contrôleur hardware)
- Systèmes critiques (avion, train, etc...)

Inconvénients

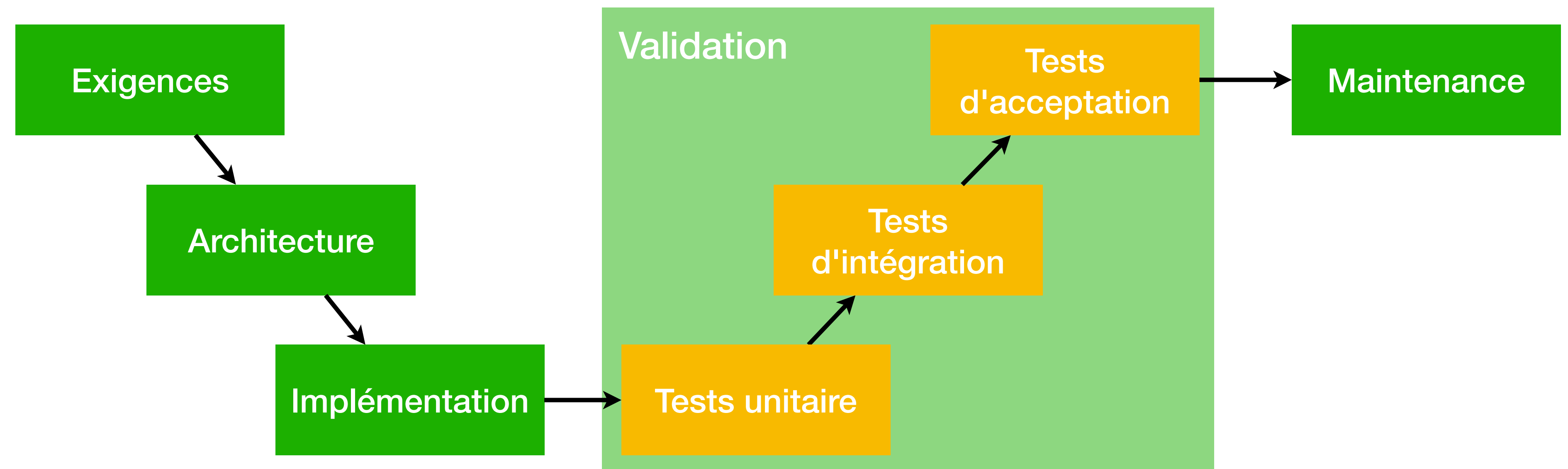
- Très peu flexible aux changements
- Parfois difficile de tout spécifier en amont
- Temps de développement conséquents



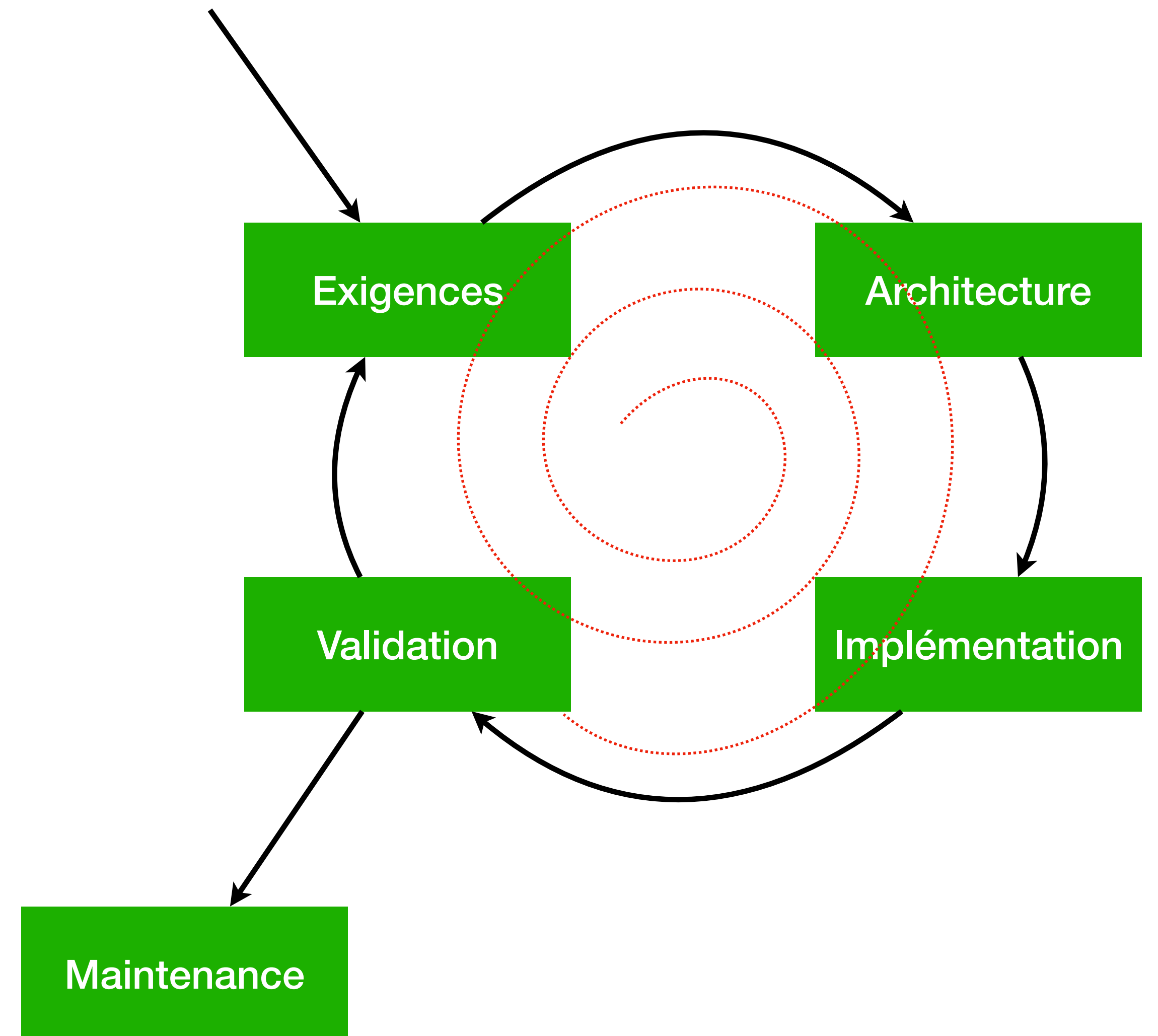
Cycle en V

Légère variation du cycle en cascade

- Met en regard chaque activité avec sa phase de validation
- Met en avant le rôle du test à tout les niveaux
- Partage les avantages et inconvénients du cycle en cascade



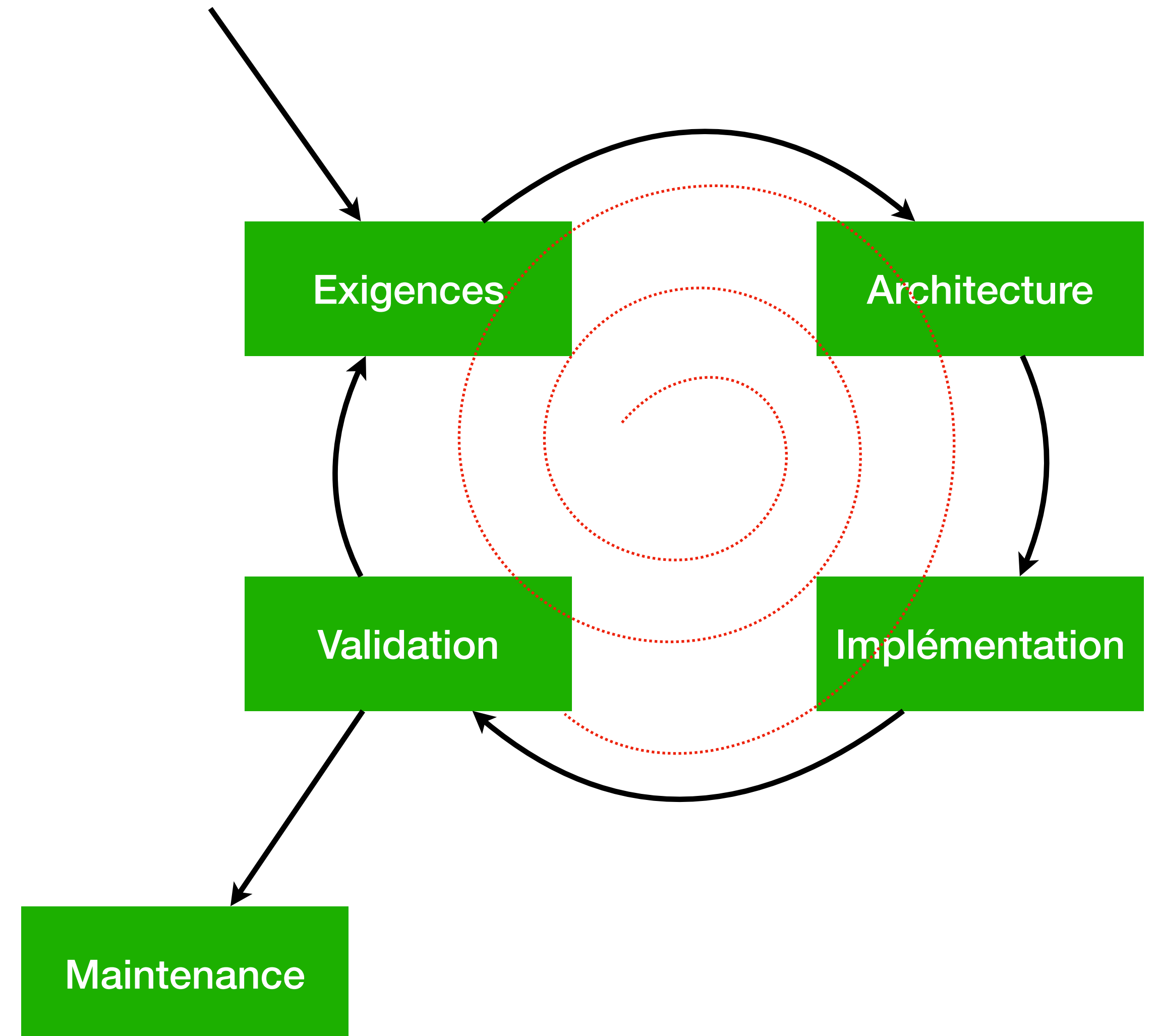
Processus itératif



Processus itératif

Logiciel évolue sur plusieurs versions

- Un prototype rapide est discuté avec l'utilisateur
- Exigences raffinées à chaque itération



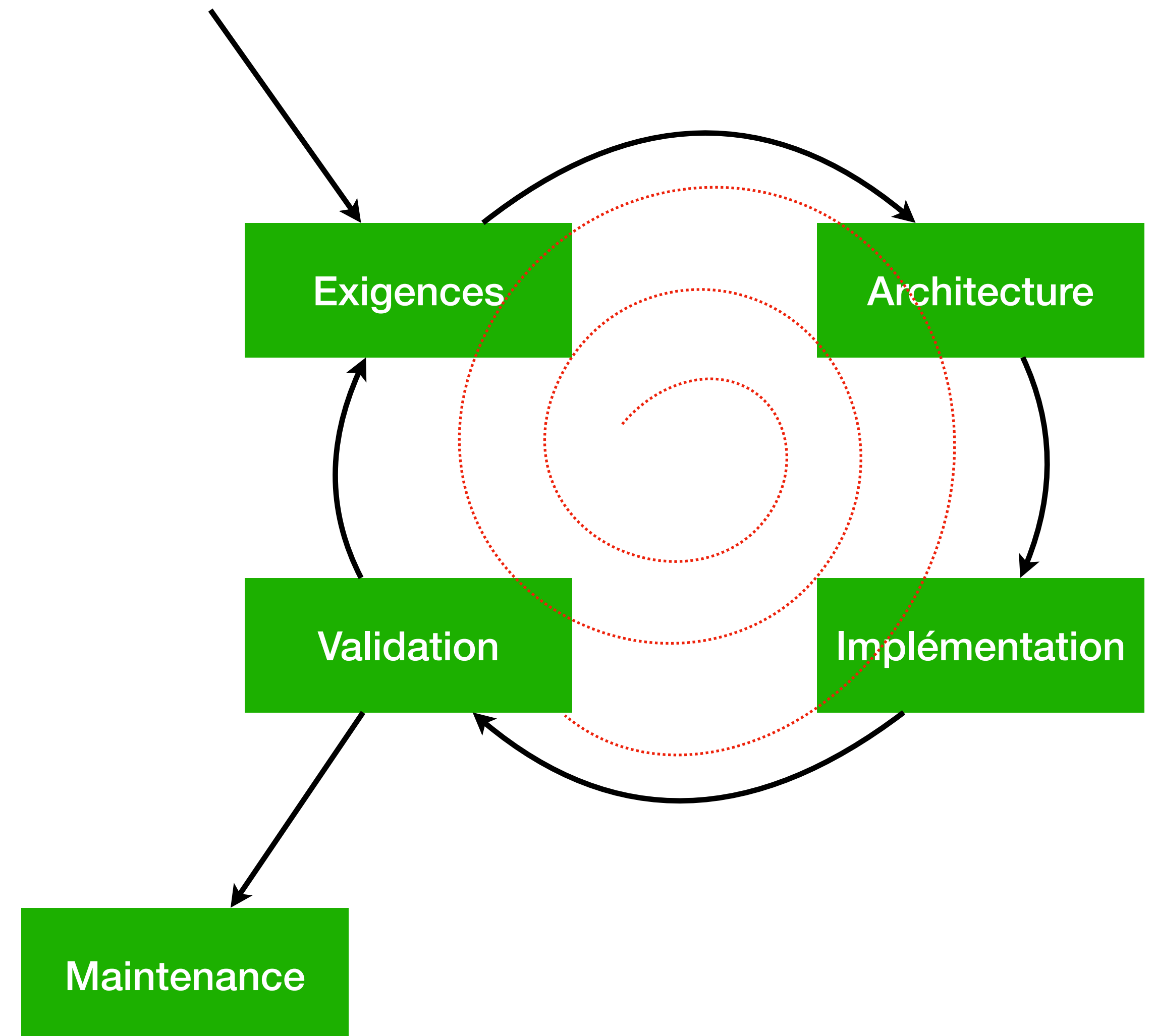
Processus itératif

Logiciel évolue sur plusieurs versions

- Un prototype rapide est discuté avec l'utilisateur
- Exigences raffinées à chaque itération

Avantages

- Retour rapide des utilisateurs
- Flexible : exigences peuvent facilement changer
- Livraison rapide



Processus itératif

Logiciel évolue sur plusieurs versions

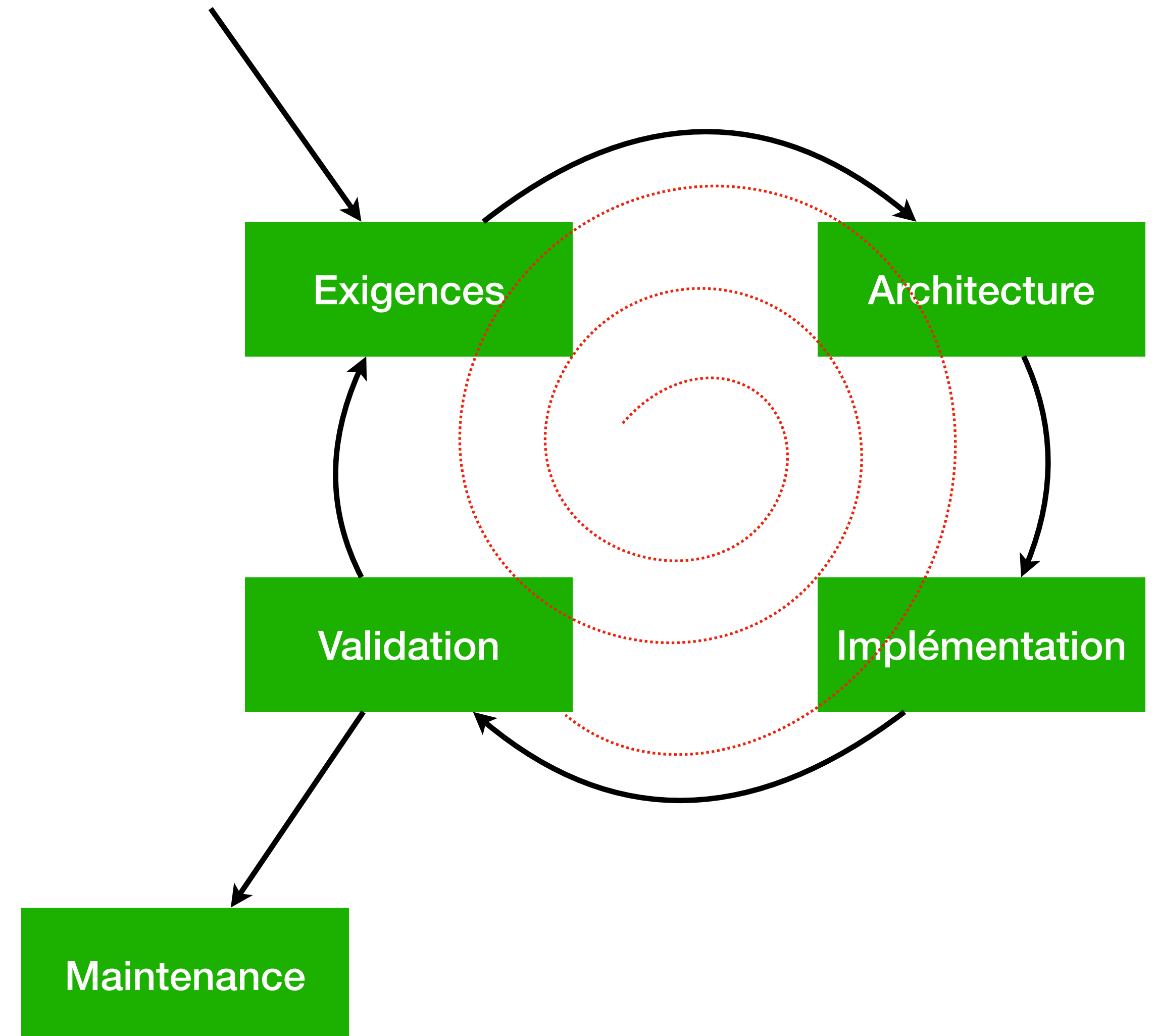
- Un prototype rapide est discuté avec l'utilisateur
- Exigences raffinées à chaque itération

Avantages

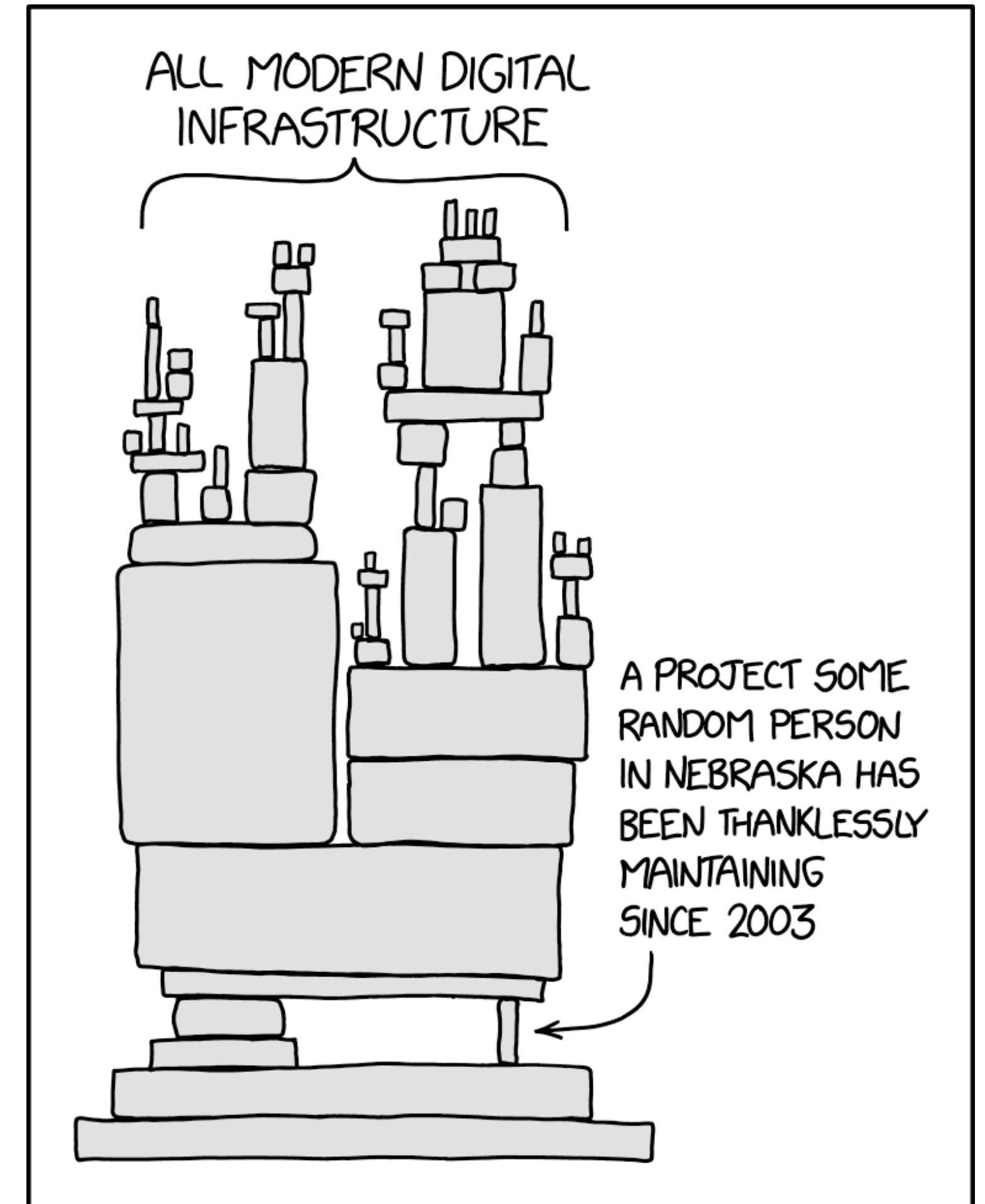
- Retour rapide des utilisateurs
- Flexible : exigences peuvent facilement changer
- Livraison rapide

Inconvénients

- Prototypes peuvent être fragiles
- Difficile de (re)-structurer le projet



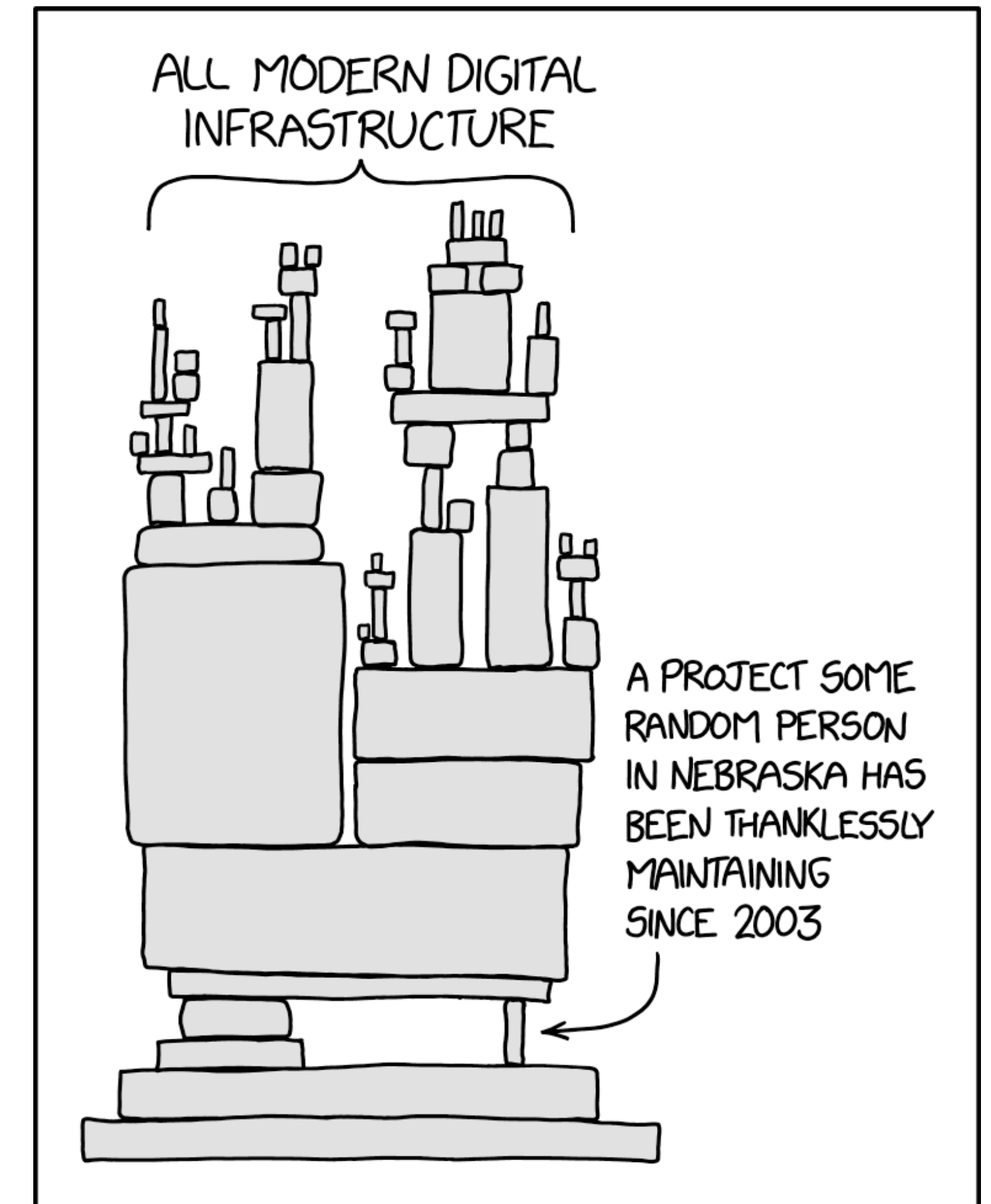
Modèle par composants



Modèle par composants

Séparation des responsabilités

- Décomposer les exigences en composants
- Réutiliser au maximum des composants existants
- Exemple : web API, microservices, ...



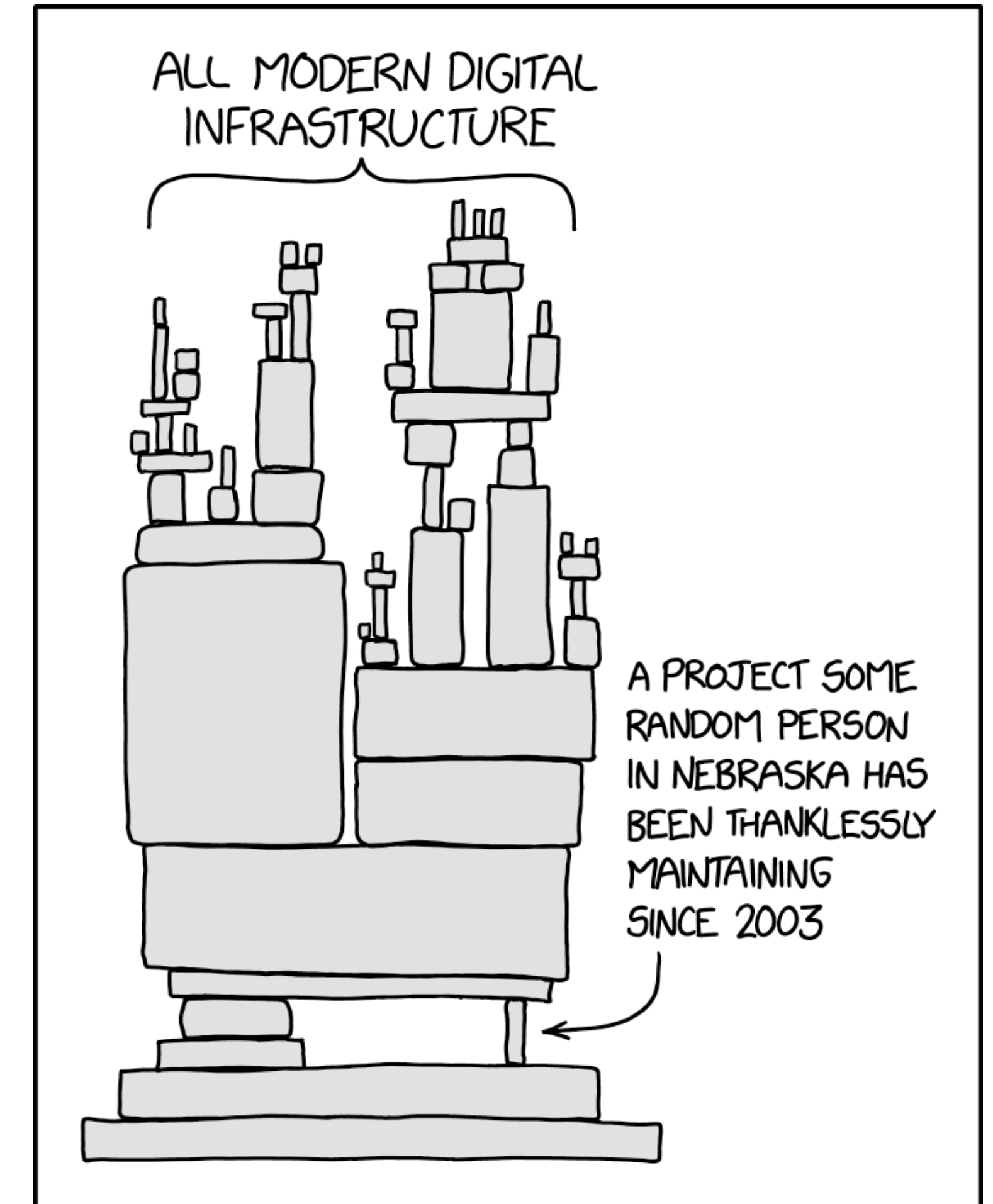
Modèle par composants

Séparation des responsabilités

- Décomposer les exigences en composants
- Réutiliser au maximum des composants existants
- Exemple : web API, microservices, ...

Avantages

- Coûts de développement réduit
- Livraison rapide



Modèle par composants

Séparation des responsabilités

- Décomposer les exigences en composants
- Réutiliser au maximum des composants existants
- Exemple : web API, microservices, ...

Avantages

- Coûts de développement réduit
- Livraison rapide

Inconvénients

- Compromis sur les exigences
- Perte de contrôle
- Coût de maintenance



Hacker News : Angular 4.0

Activités

Génie logiciel

Exigences

Architecture

Implémentation

Validation

Maintenance

Exigences

Exigences

Exigences fonctionnelles

- Que doit faire le système ?
- Comment le système réagit ?
- Ce que le système ne doit pas faire ?

Exigences

Exigences fonctionnelles

- Que doit faire le système ?
- Comment le système réagit ?
- Ce que le système ne doit pas faire ?



Exigences

Exigences fonctionnelles

- Que doit faire le système ?
- Comment le système réagit ?
- Ce que le système ne doit pas faire ?



En cas d'infection les cas contact doivent être prévenu

Le système ne doit pas révéler l'identité des malades aux autres utilisateurs

Un utilisateur doit être identifié par un numéro unique

Exigences



Exigences fonctionnelles

- Que doit faire le système ?
- Comment le système réagit ?
- Ce que le système ne doit pas faire ?

Exigence non-fonctionnelles

- Contraintes sur le système entier
- Comment le système doit être ?
- Exemples :
performances, mémoire, robustesse, sécurité, sûreté, capacité, gestions des données, temps de réponse, open-source, licence, environnement, ...

En cas d'infection les cas contact doivent être prévenu

Le système ne doit pas révéler l'identité des malades aux autres utilisateurs

Un utilisateur doit être identifié par un numéro unique

Exigences



Exigences fonctionnelles

- Que doit faire le système ?
- Comment le système réagit ?
- Ce que le système ne doit pas faire ?

Exigence non-fonctionnelles

- Contraintes sur le système entier
- Comment le système doit être ?
- Exemples :
performances, mémoire, robustesse, sécurité, sûreté, capacité, gestions des données, temps de réponse, open-source, licence, environnement, ...

En cas d'infection les cas contact doivent être prévenu

Le système ne doit pas révéler l'identité des malades aux autres utilisateurs

Un utilisateur doit être identifié par un numéro unique

Le système doit pouvoir être utilisable par 60+M d'utilisateurs

Le système doit pouvoir être facilement étendu pour s'adapter aux nouvelles mesures (traçage, QR codes)

Le système doit stocker le minimum de données personnelle sur le serveur (GDPR)

Exigences

Exigences

Difficultés

- Ambiguïtés : exigences précises
- Complétude : couvrir toutes les fonctionnalités
- Coherence : éviter les contradictions
- Niveau de détails : formel, semi-formel, informel

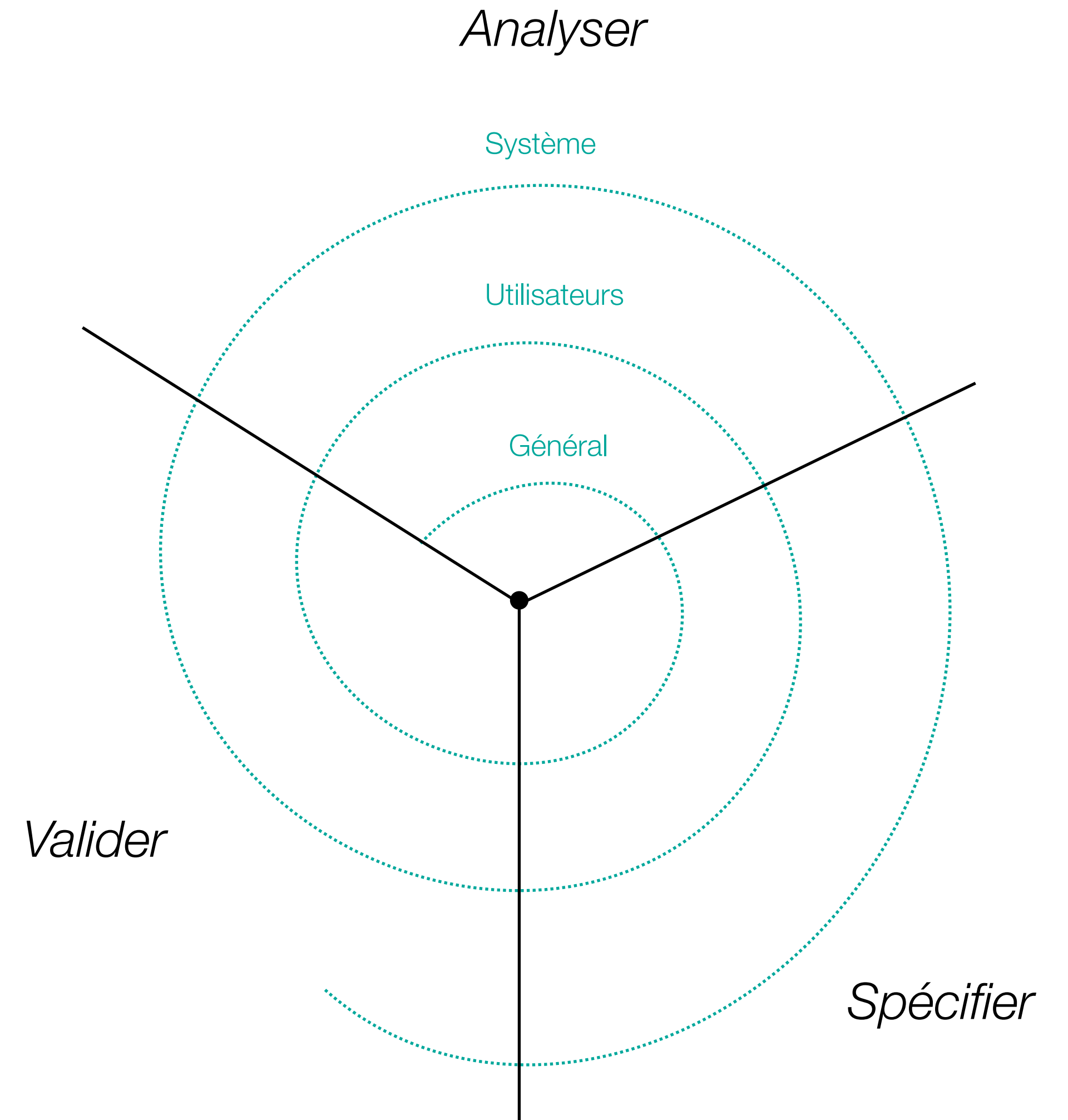
Exigences

Difficultés

- Ambiguïtés : exigences précises
- Complétude : couvrir toutes les fonctionnalités
- Coherence : éviter les contradictions
- Niveau de détails : formel, semi-formel, informel

Processus de développement des exigences

- Analyser: identifier, collecter, déduire des exigences techniques
- Spécifier : documenter sous une forme standard
- Valider : vérifier la cohérence, revue



Architecture

Architecture

Concevoir une architecture logicielle pour réaliser les spécifications

- Définitions des composants
- Définition des interfaces
- Orchestration des composants

Architecture

Concevoir une architecture logicielle pour réaliser les spécifications

- Définitions des composants
- Définition des interfaces
- Orchestration des composants

Plusieurs objectifs, plusieurs architectures possibles

- Performance
- Sécurité
- Sûreté
- Maintenance
- ...

Architecture

Concevoir une architecture logicielle pour réaliser les spécifications

- Définitions des composants
- Définition des interfaces
- Orchestration des composants

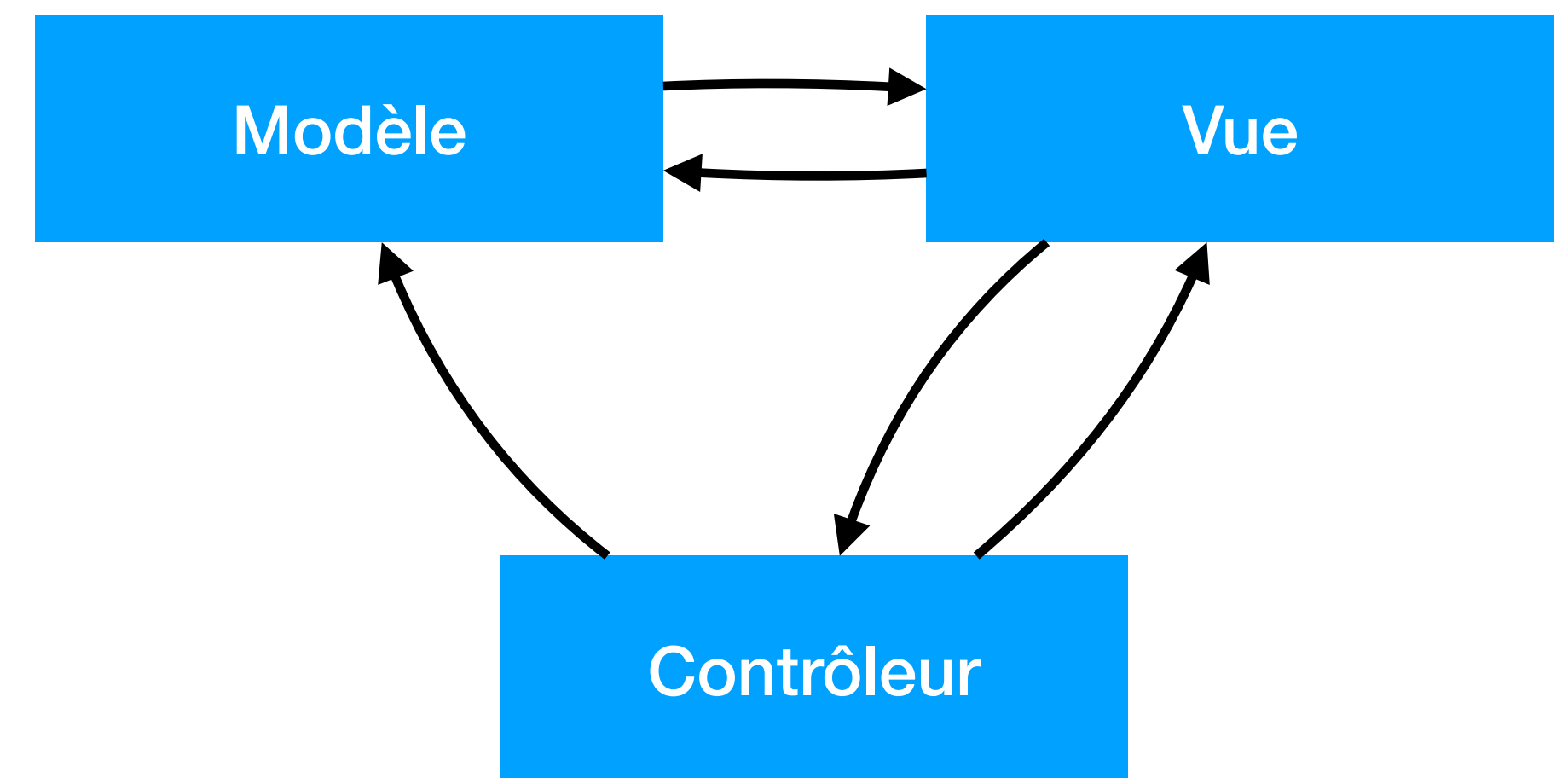
Plusieurs objectifs, plusieurs architectures possibles

- Performance
- Sécurité
- Sûreté
- Maintenance
- ...

Plusieurs descriptions possibles

- Simples diagrammes (sans sémantique)
- Diagrammes UML
- Plusieurs points de vue possibles (logique, physique, développement, processus)

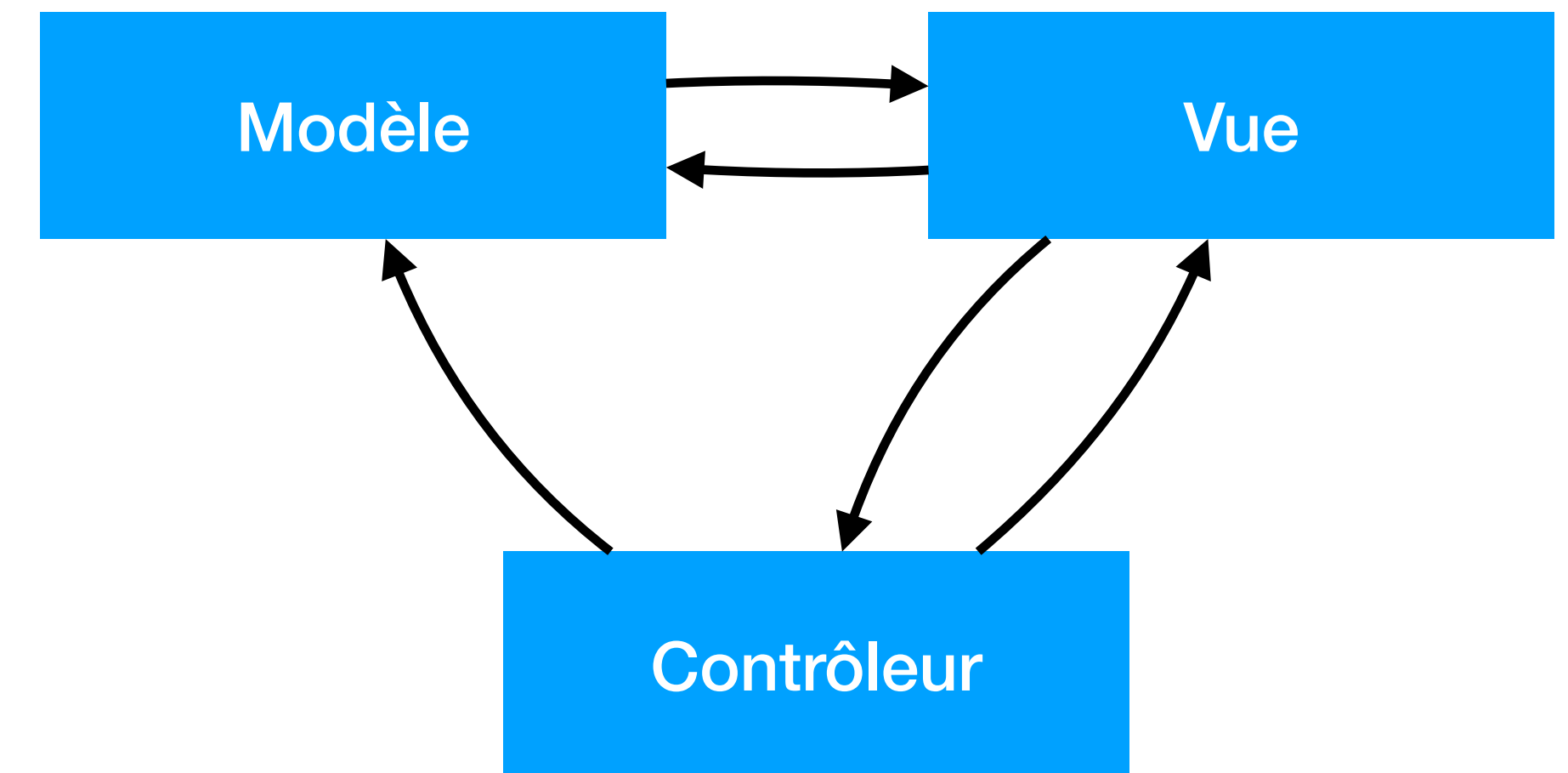
Architecture MVC



Architecture MVC

Modèle/Vue/Contrôleur

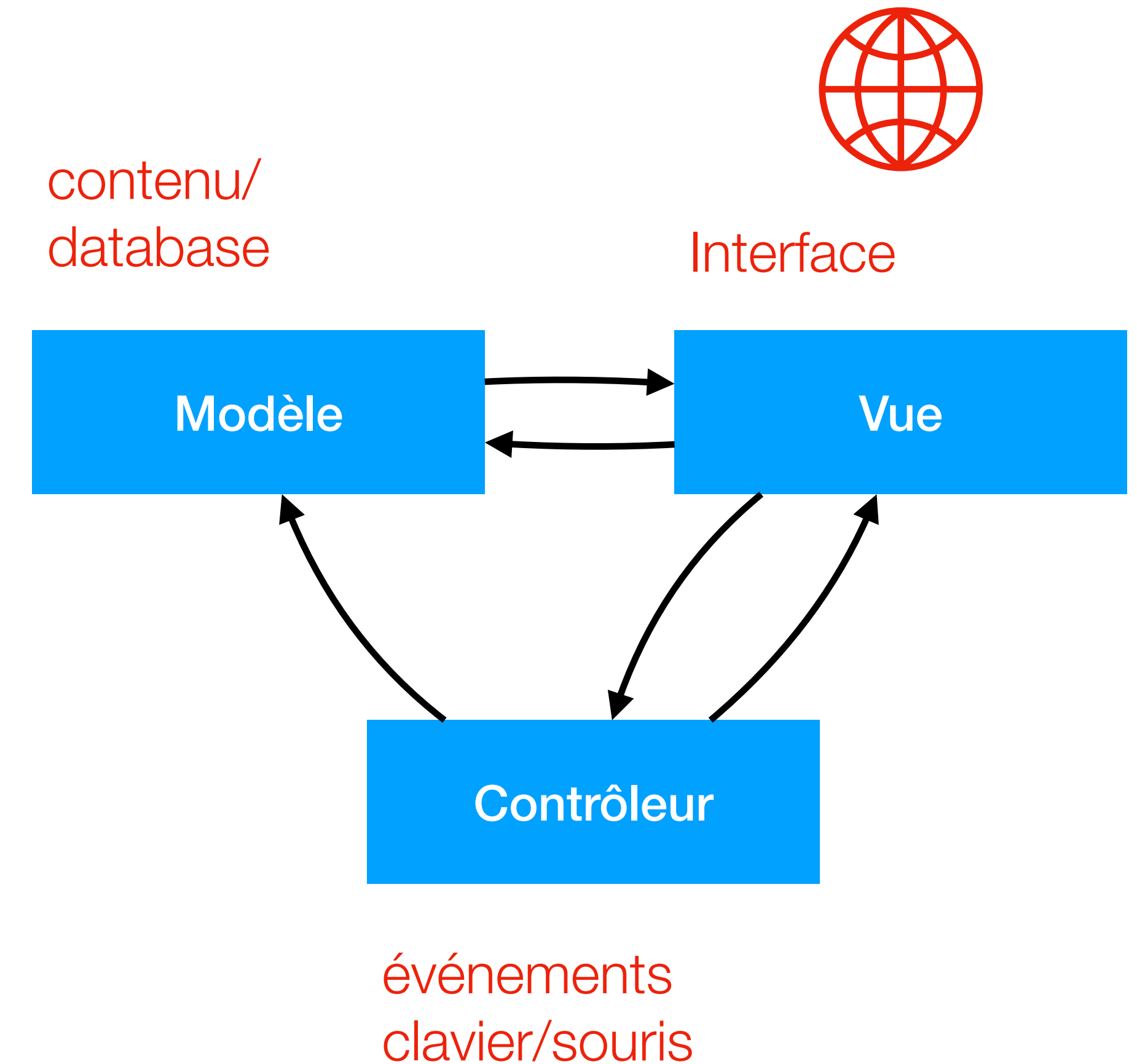
- Le modèle contient les données
- La vue contient la présentation (interface graphique)
- Le contrôleur gère la logique
- Exemple : Interface graphique, applications web



Architecture MVC

Modèle/Vue/Contrôleur

- Le modèle contient les données
- La vue contient la présentation (interface graphique)
- Le contrôleur gère la logique
- Exemple : Interface graphique, applications web



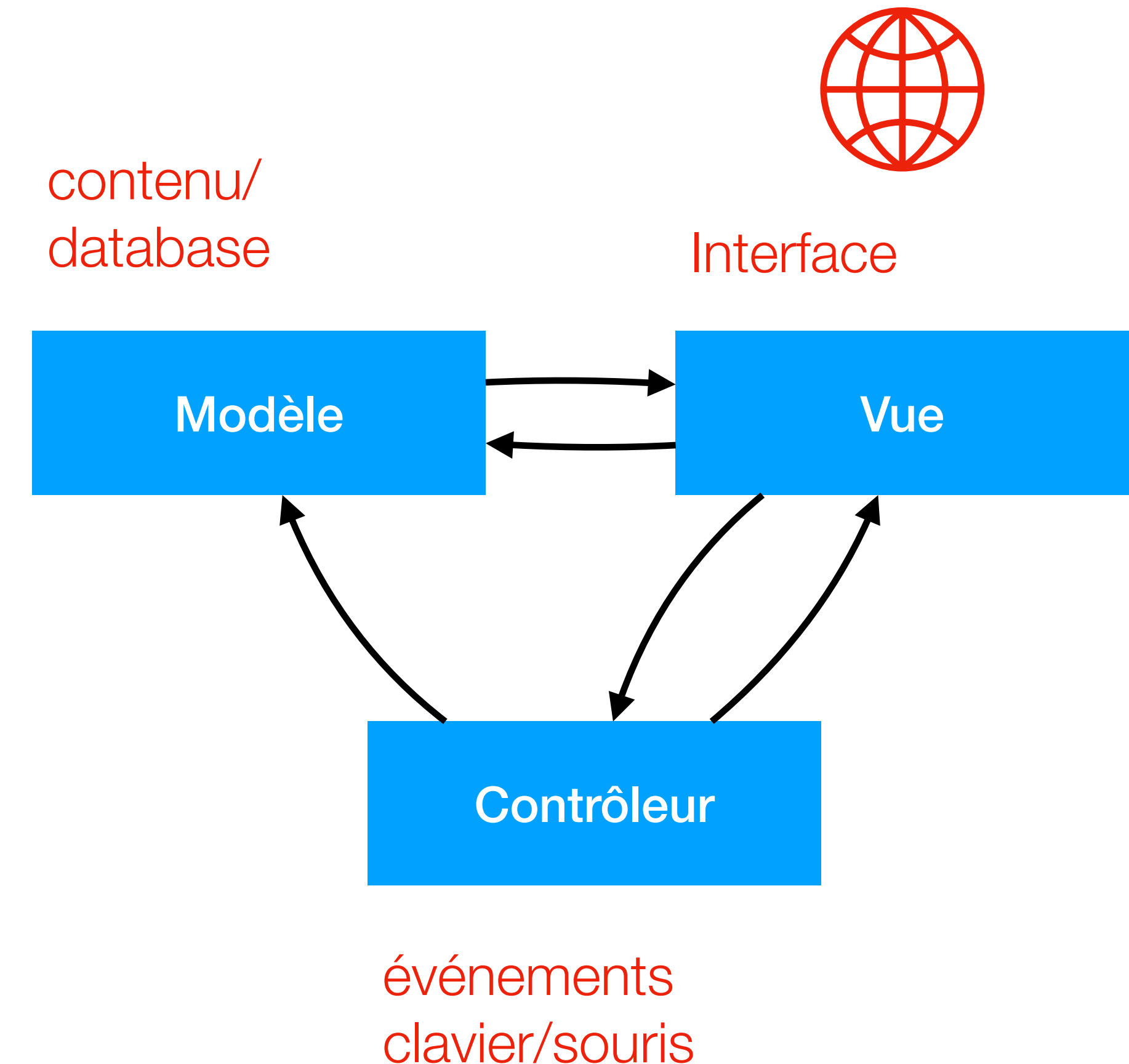
Architecture MVC

Modèle/Vue/Contrôleur

- Le modèle contient les données
- La vue contient la présentation (interface graphique)
- Le contrôleur gère la logique
- Exemple : Interface graphique, applications web

Avantage

- Séparation données / présentation
- Plusieurs présentations possibles des mêmes données
- Architecture asynchrone



Architecture MVC

Modèle/Vue/Contrôleur

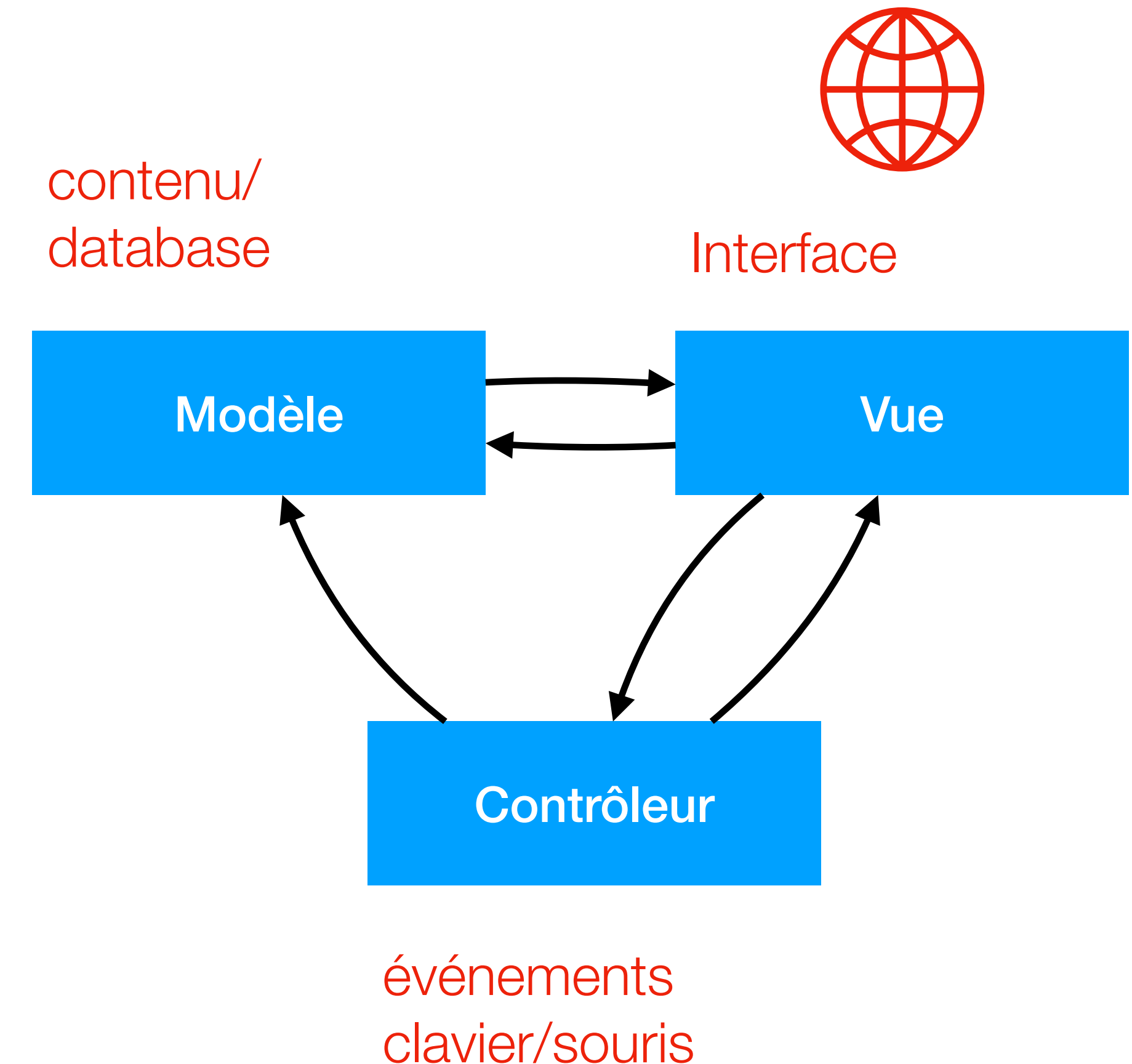
- Le modèle contient les données
- La vue contient la présentation (interface graphique)
- Le contrôleur gère la logique
- Exemple : Interface graphique, applications web

Avantage

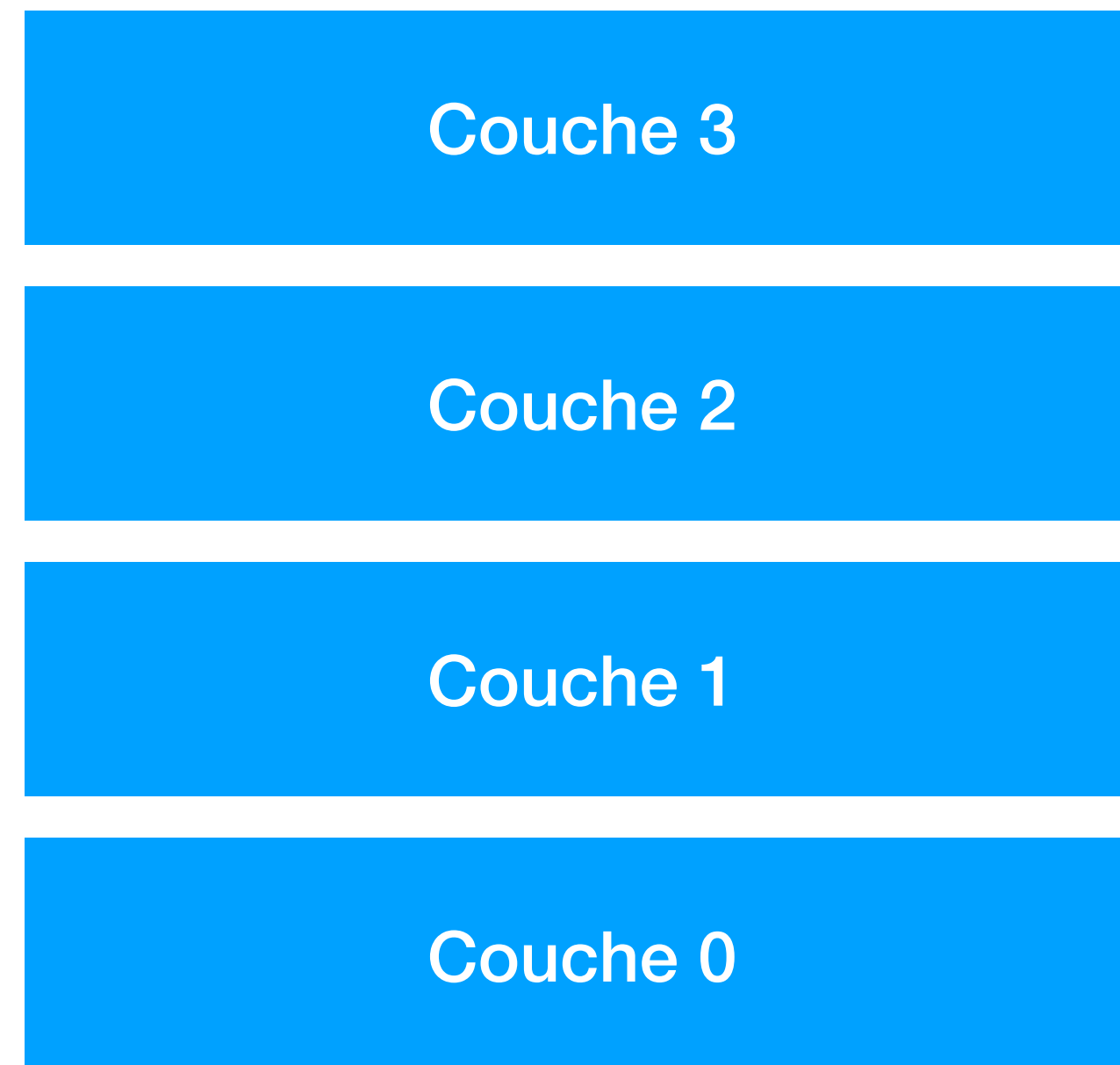
- Séparation données / présentation
- Plusieurs présentations possibles des mêmes données
- Architecture asynchrone

Inconvénients

- Code parfois inutilement compliqué pour les petites applications
- Mises-à-jour fréquentes



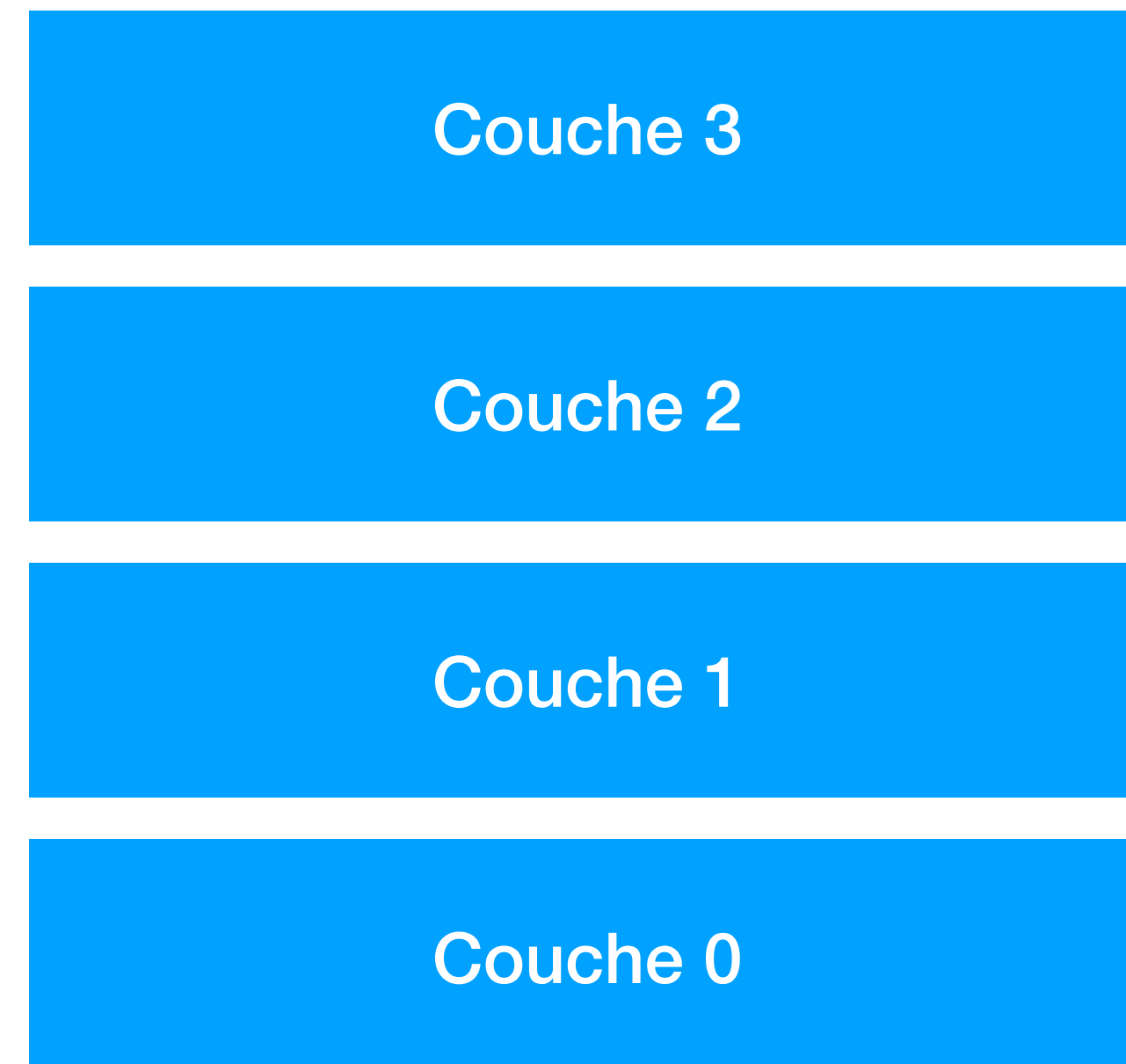
Architecture stratifiée



Architecture stratifiée

Logiciel organisé sous forme de couches superposées

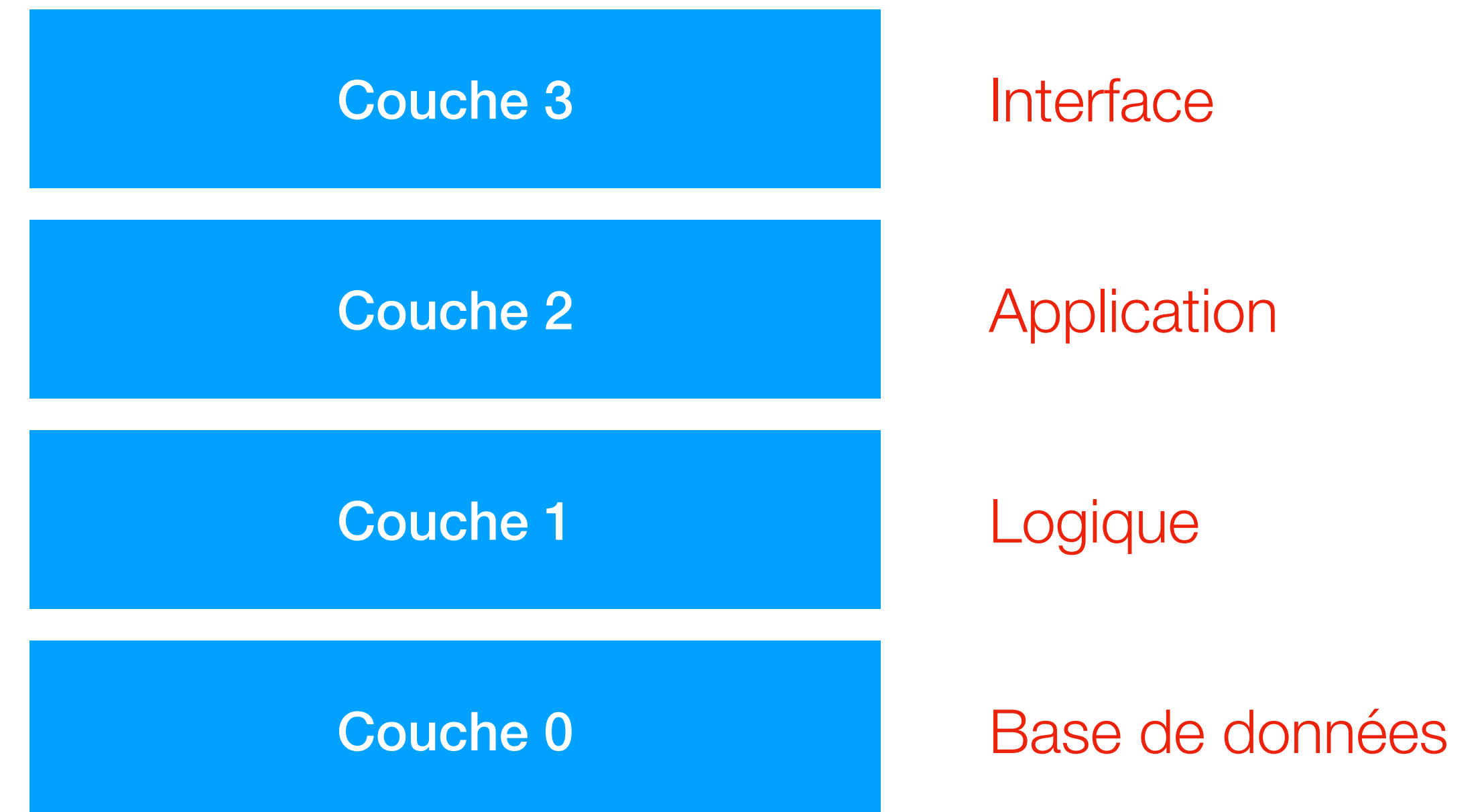
- Chaque couche correspond à une fonctionnalité
- Une couche ne peut communiquer qu'avec les couches adjacentes
- Exemples : système d'exploitation, réseaux OSI.



Architecture stratifiée

Logiciel organisé sous forme de couches superposées

- Chaque couche correspond à une fonctionnalité
- Une couche ne peut communiquer qu'avec les couches adjacentes
- Exemples : système d'exploitation, réseaux OSI.



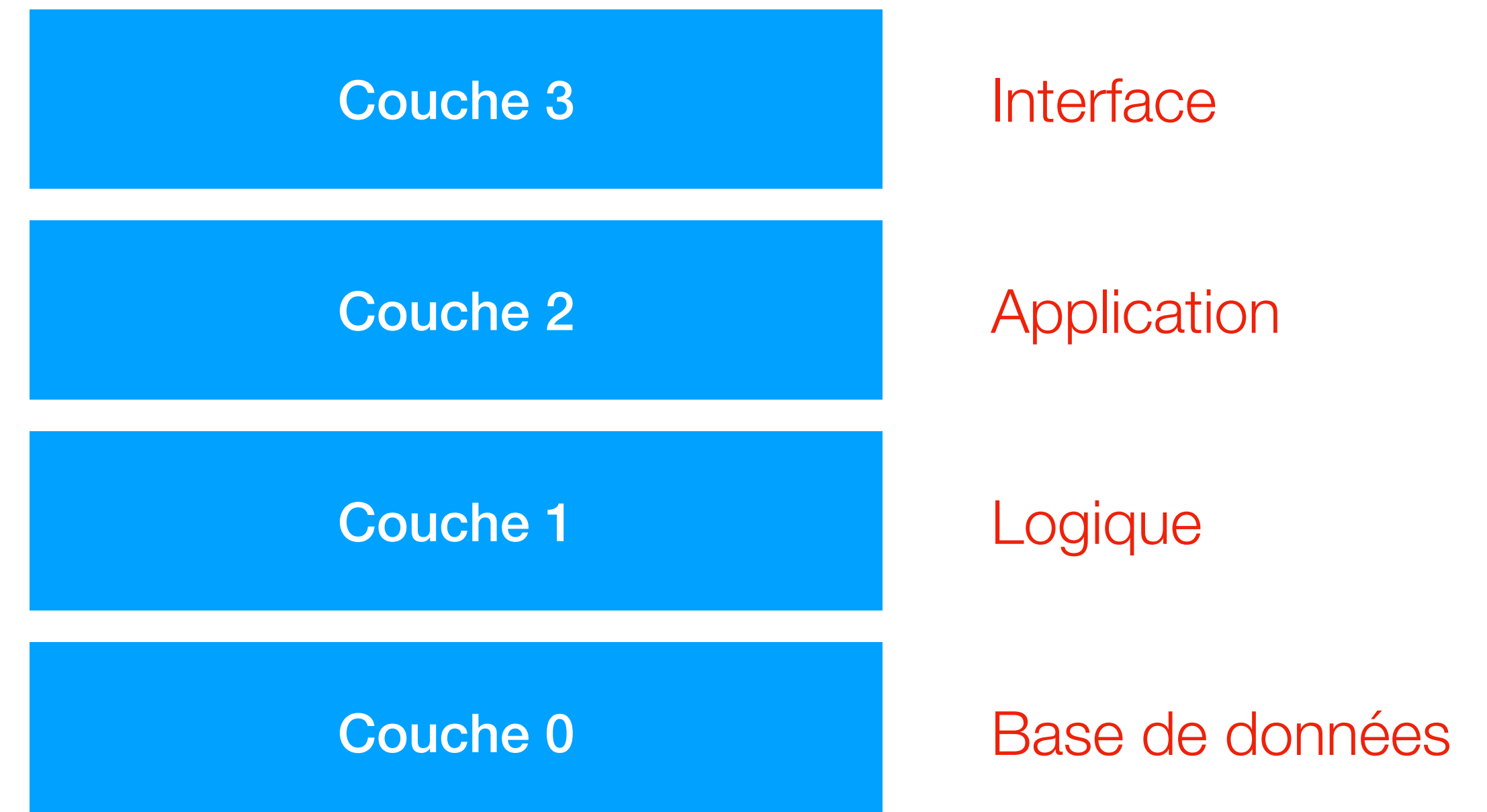
Architecture stratifiée

Logiciel organisé sous forme de couches superposées

- Chaque couche correspond à une fonctionnalité
- Une couche ne peut communiquer qu'avec les couches adjacentes
- Exemples : système d'exploitation, réseaux OSI.

Avantages

- Séparation des responsabilités
- Flexibilité, sécurité (remplacer une couche)
- Redondance, robustesse



Architecture stratifiée

Logiciel organisé sous forme de couches superposées

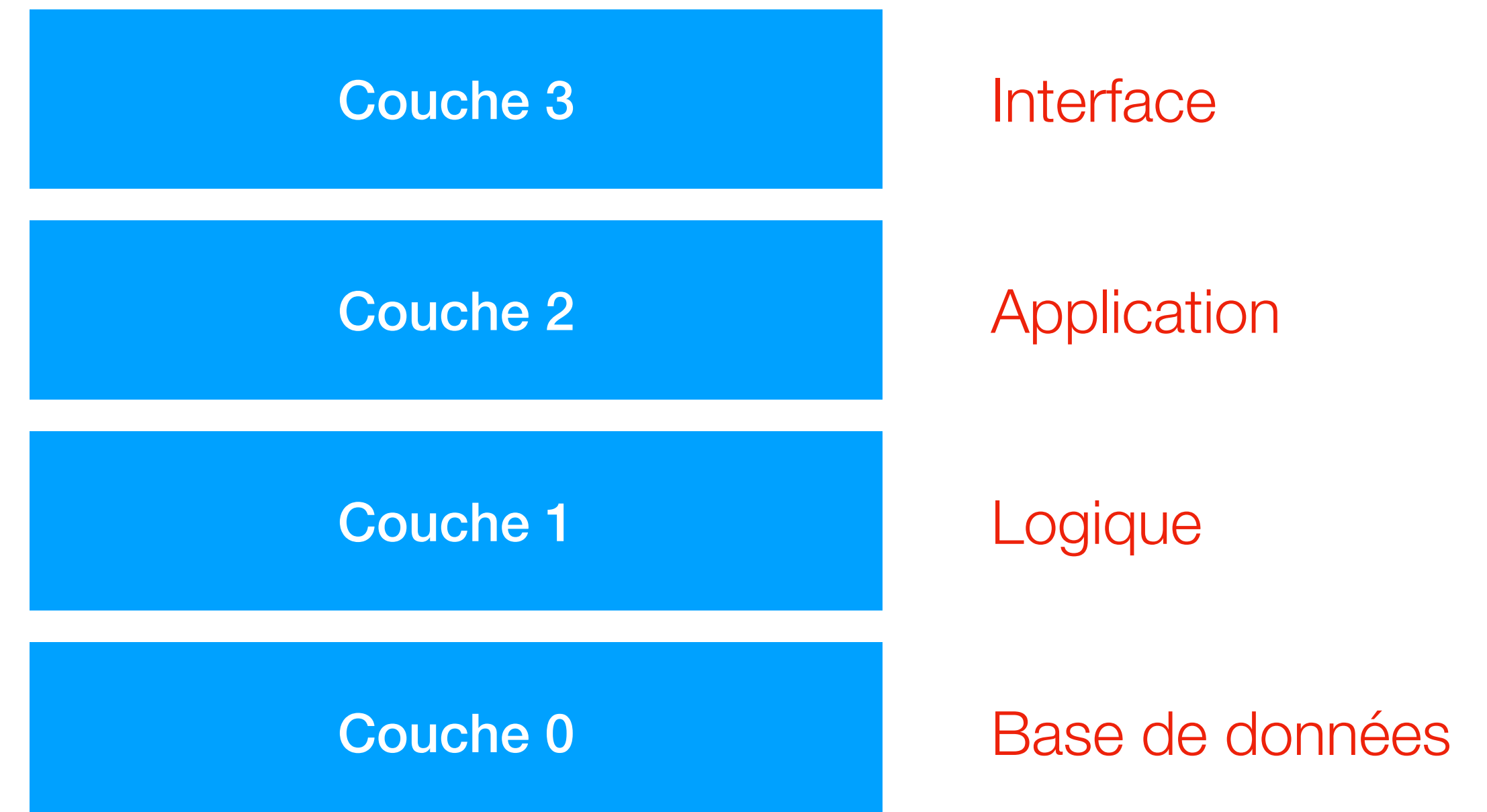
- Chaque couche correspond à une fonctionnalité
- Une couche ne peut communiquer qu'avec les couches adjacentes
- Exemples : système d'exploitation, réseaux OSI.

Avantages

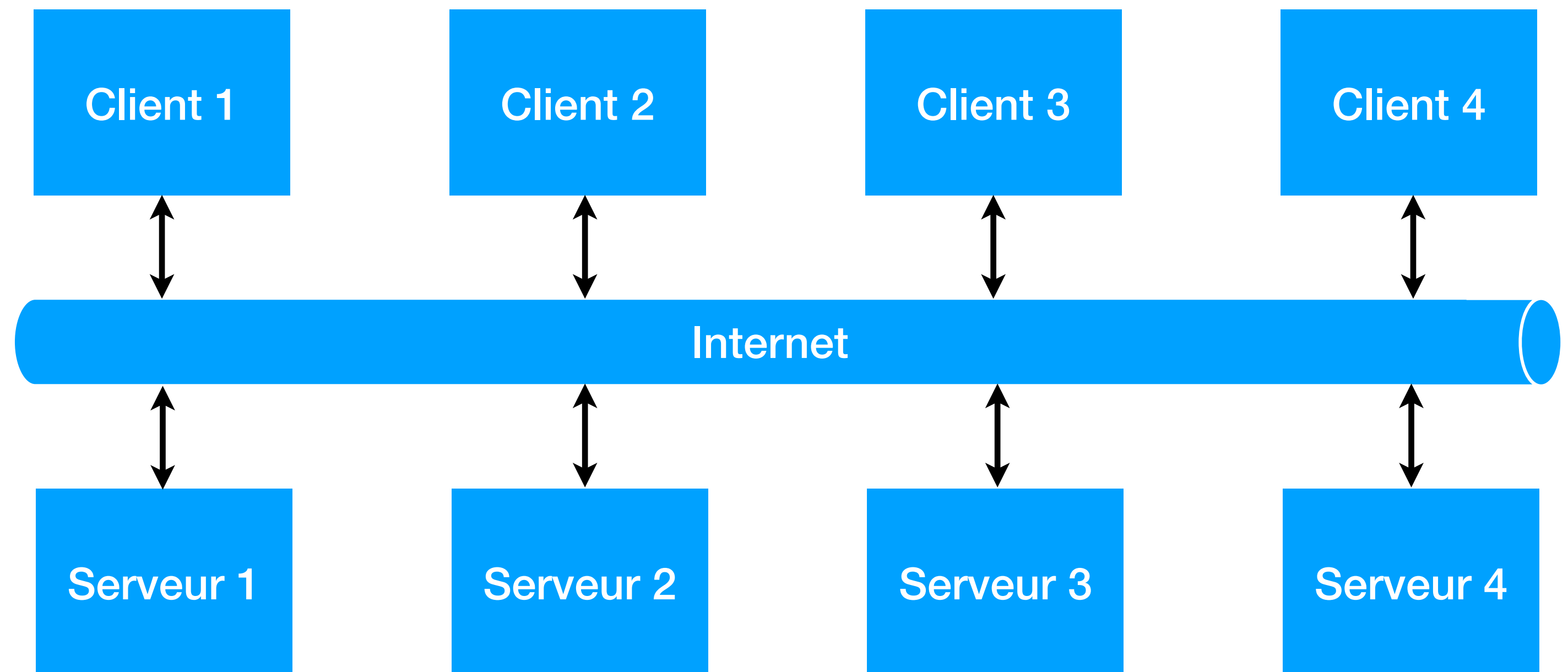
- Séparation des responsabilités
- Flexibilité, sécurité (remplacer une couche)
- Redondance, robustesse

Inconvénients

- Parfois artificielle
- Manque de souplesse
- Redondance



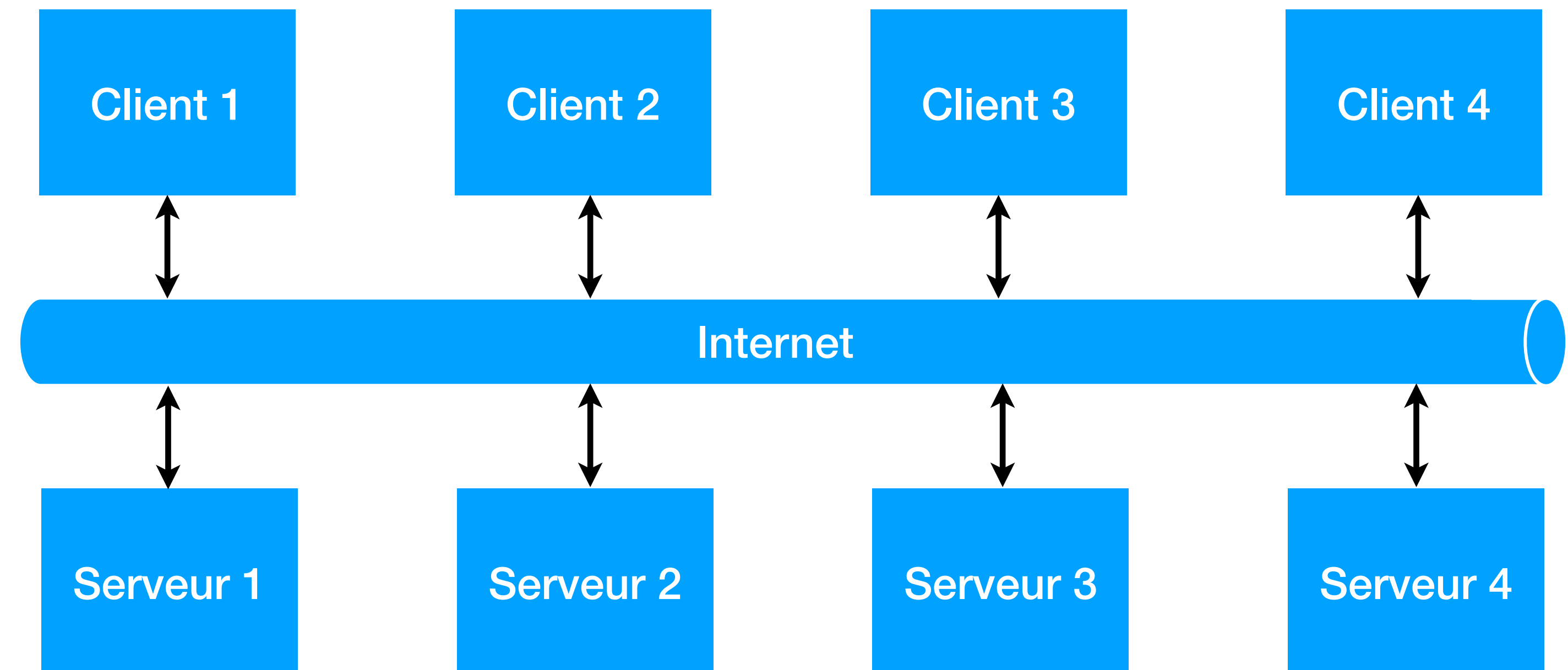
Architecture client/serveur



Architecture client/serveur

Ensemble de clients accèdent à un ensemble de services

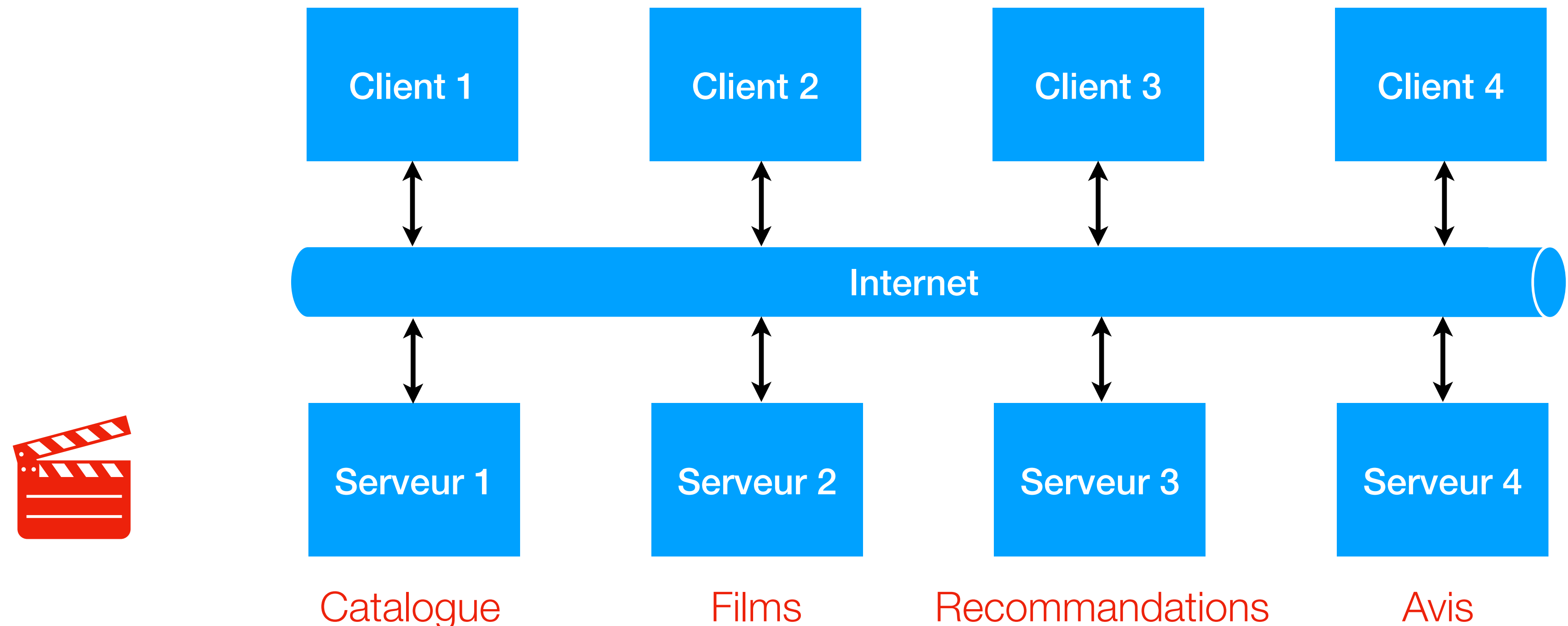
- Serveurs offrent des services (système de fichiers, compilation, execution)
- Plusieurs clients appellent les services offerts par les serveurs
- La communication est réalisée par le réseau
- Exemple : email, web,



Architecture client/serveur

Ensemble de clients accèdent à un ensemble de services

- Serveurs offrent des services (système de fichiers, compilation, execution)
- Plusieurs clients appellent les services offerts par les serveurs
- La communication est réalisée par le réseau
- Exemple : email, web,



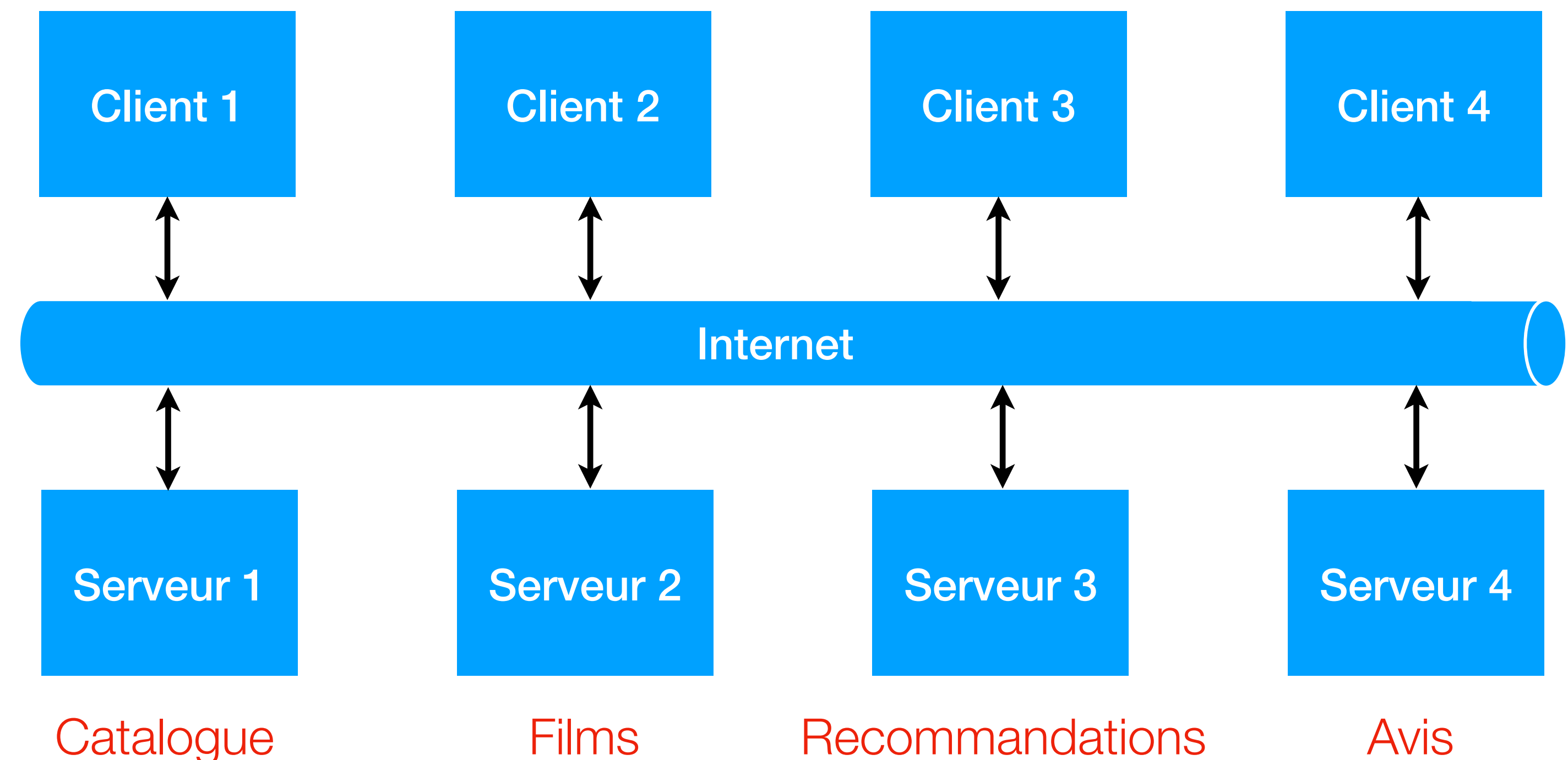
Architecture client/serveur

Ensemble de clients accèdent à un ensemble de services

- Serveurs offrent des services (système de fichiers, compilation, execution)
- Plusieurs clients appellent les services offerts par les serveurs
- La communication est réalisée par le réseau
- Exemple : email, web,

Avantages

- Architecture distribuée
- Passage à l'échelle



Architecture client/serveur

Ensemble de clients accèdent à un ensemble de services

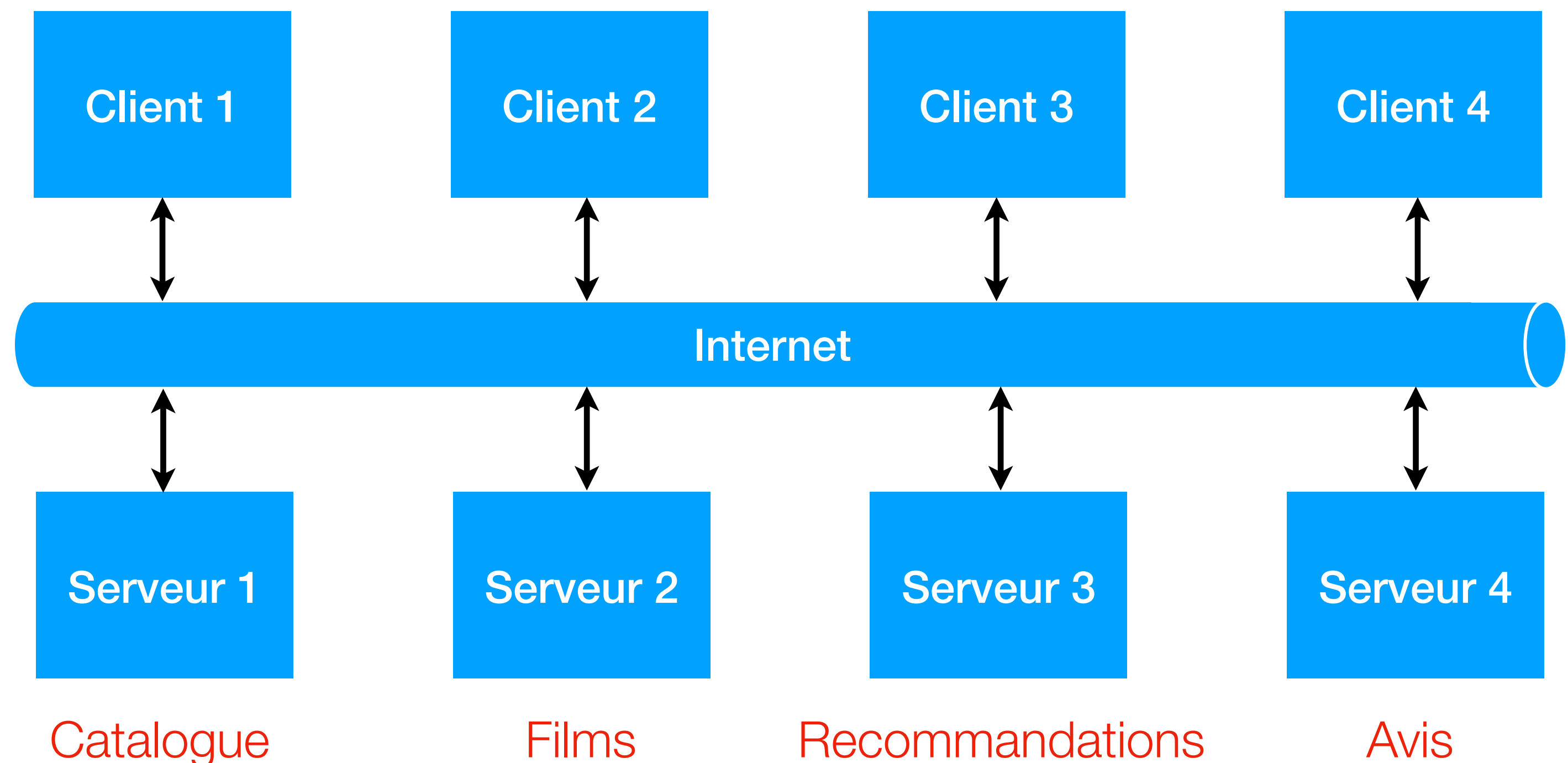
- Serveurs offrent des services (système de fichiers, compilation, execution)
- Plusieurs clients appellent les services offerts par les serveurs
- La communication est réalisée par le réseau
- Exemple : email, web,

Avantages

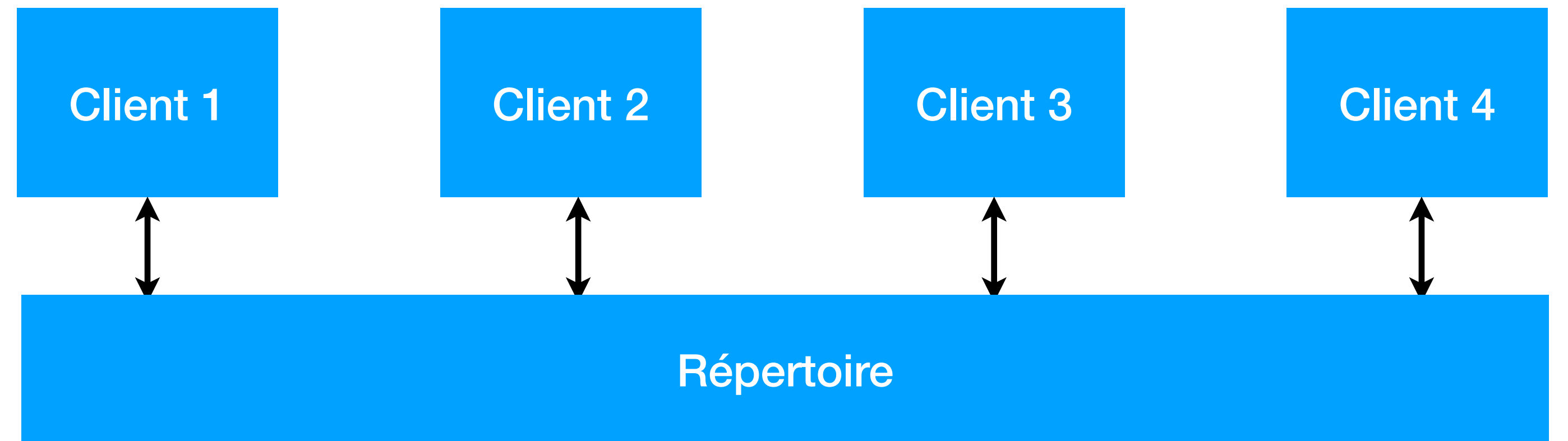
- Architecture distribuée
- Passage à l'échelle

Inconvénients

- Vulnérabilité de chaque service
- Dépendant du réseau



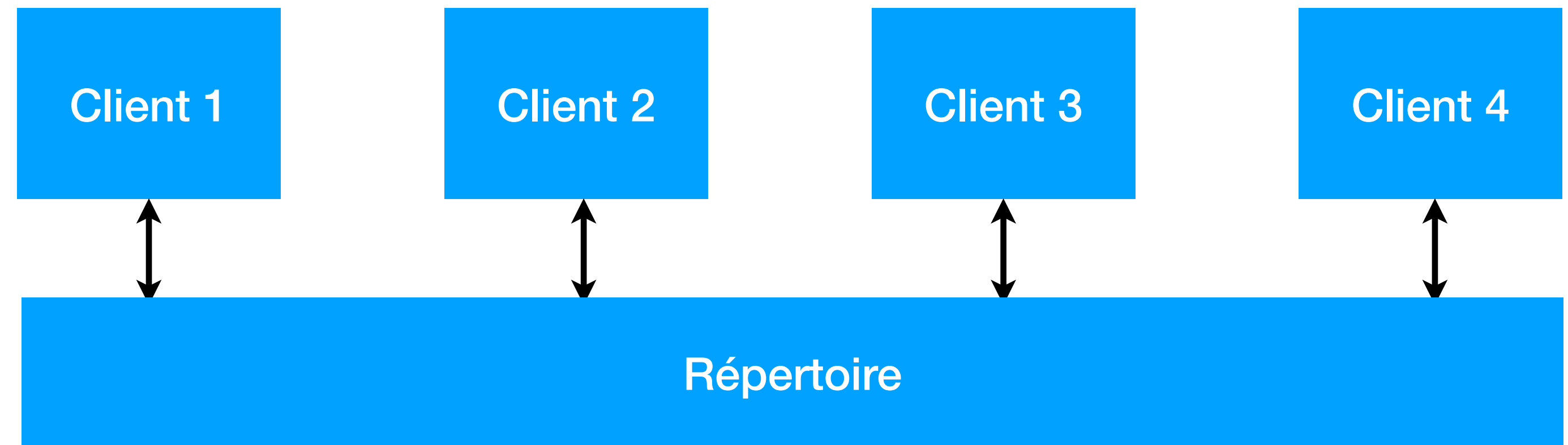
Architecture répertoire



Architecture répertoire

Répertoire central contenant les données

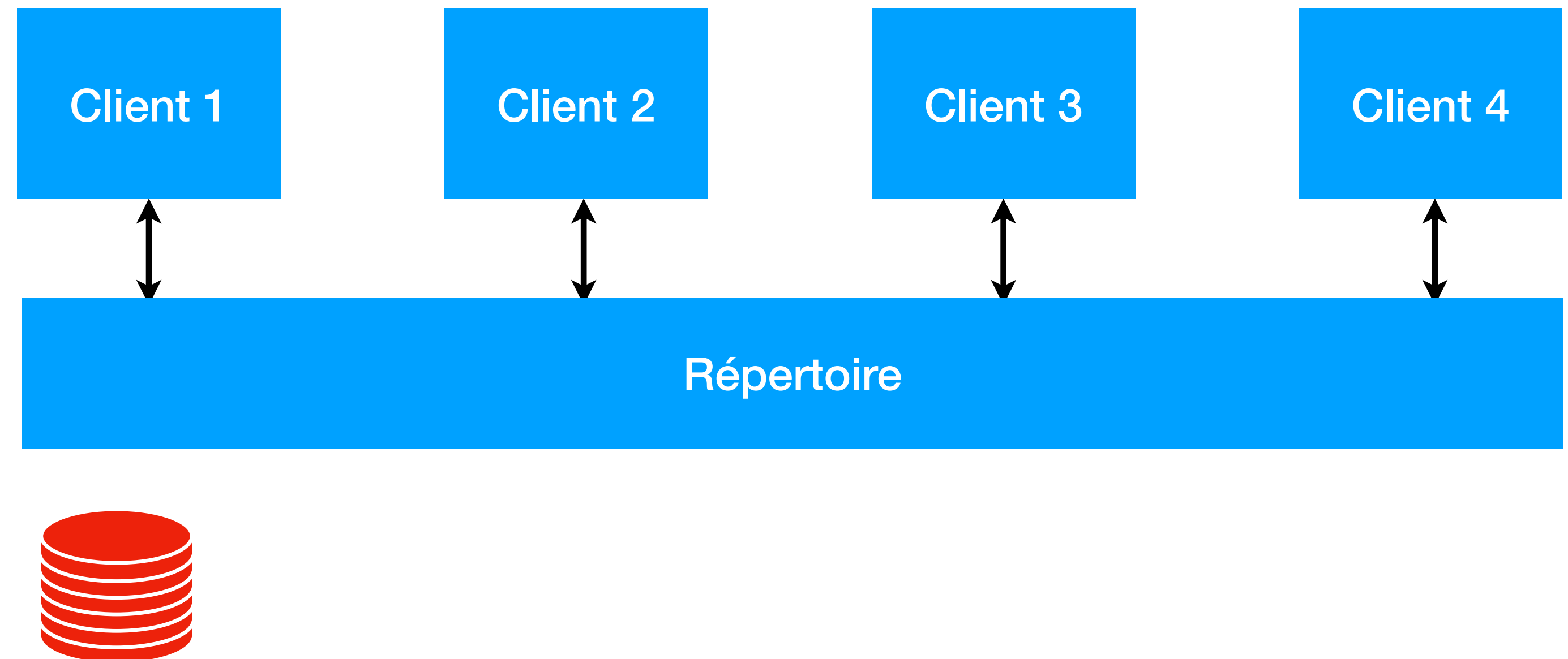
- Données accessibles à tous les composants
- Pas d'interaction directe entre composants
- Exemple : base de données



Architecture répertoire

Répertoire central contenant les données

- Données accessibles à tous les composants
- Pas d'interaction directe entre composants
- Exemple : base de données



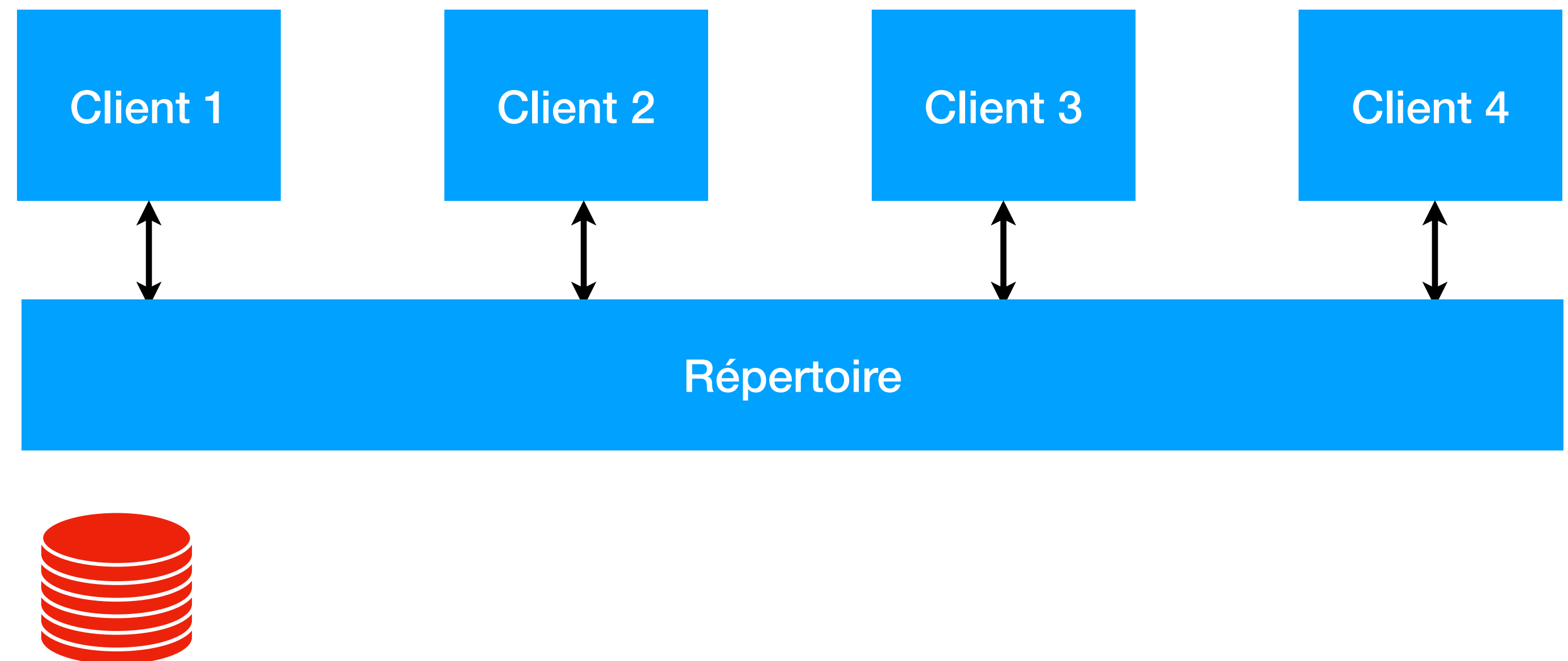
Architecture répertoire

Répertoire central contenant les données

- Données accessibles à tous les composants
- Pas d'interaction directe entre composants
- Exemple : base de données

Avantages

- Robustesse (pannes clients)
- Une seule dépendance (les données)
- Passage à l'échelle
- Flexible



Architecture répertoire

Répertoire central contenant les données

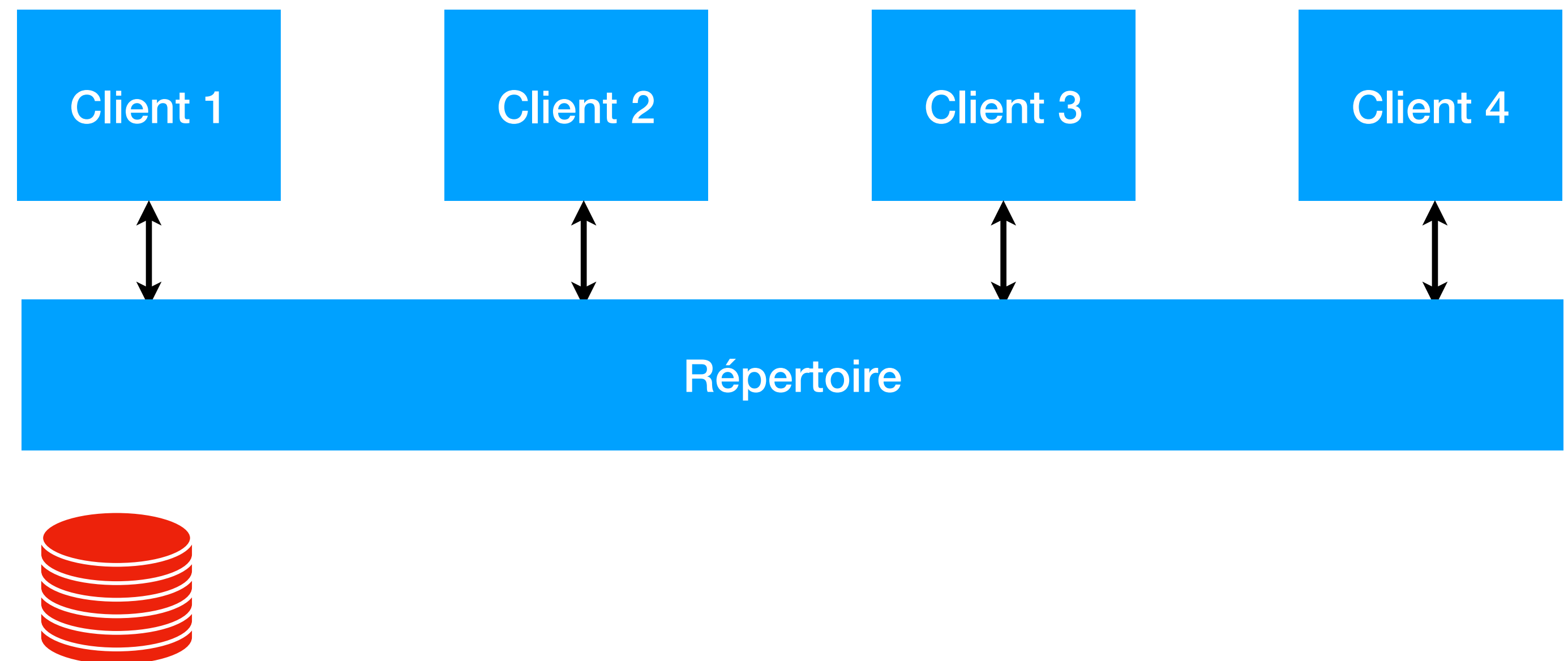
- Données accessibles à tous les composants
- Pas d'interaction directe entre composants
- Exemple : base de données

Avantages

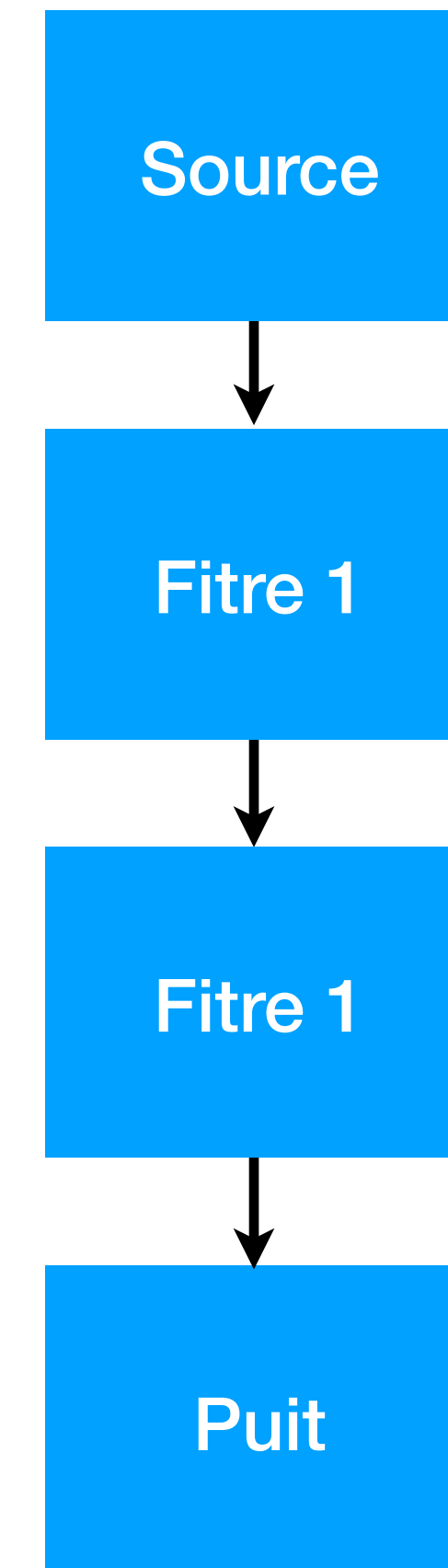
- Robustesse (pannes clients)
- Une seule dépendance (les données)
- Passage à l'échelle
- Flexible

Inconvénients

- Interblocage, famine
- Une seule dépendance (les données)



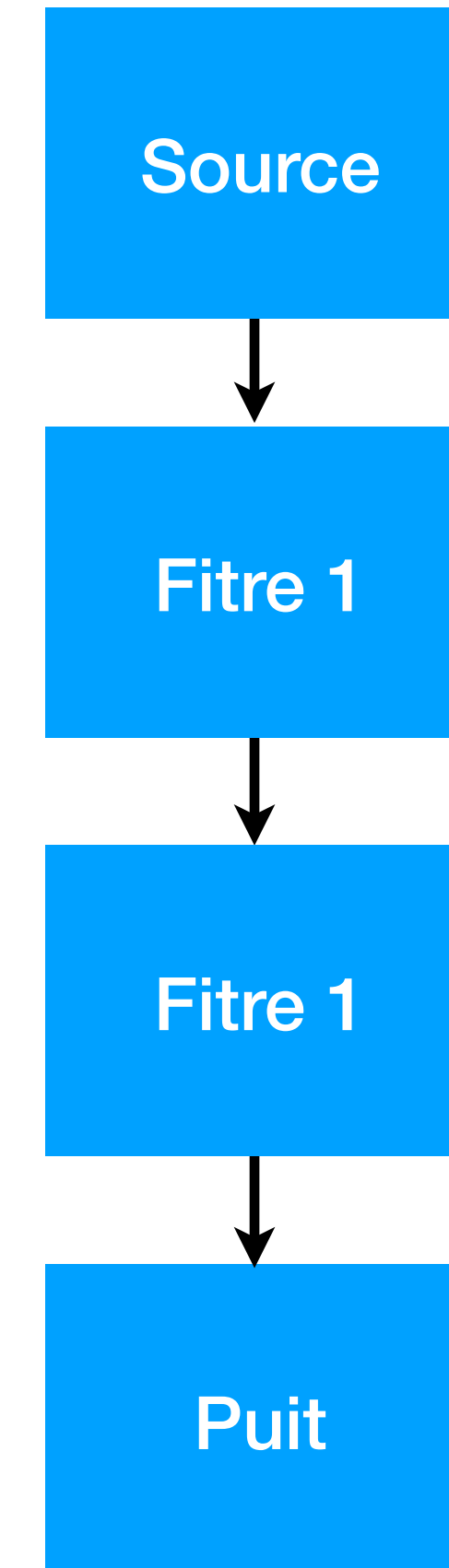
Architecture tuyaux et filtres



Architecture tuyaux et filtres

Chaque composant applique une transformation à une entrée

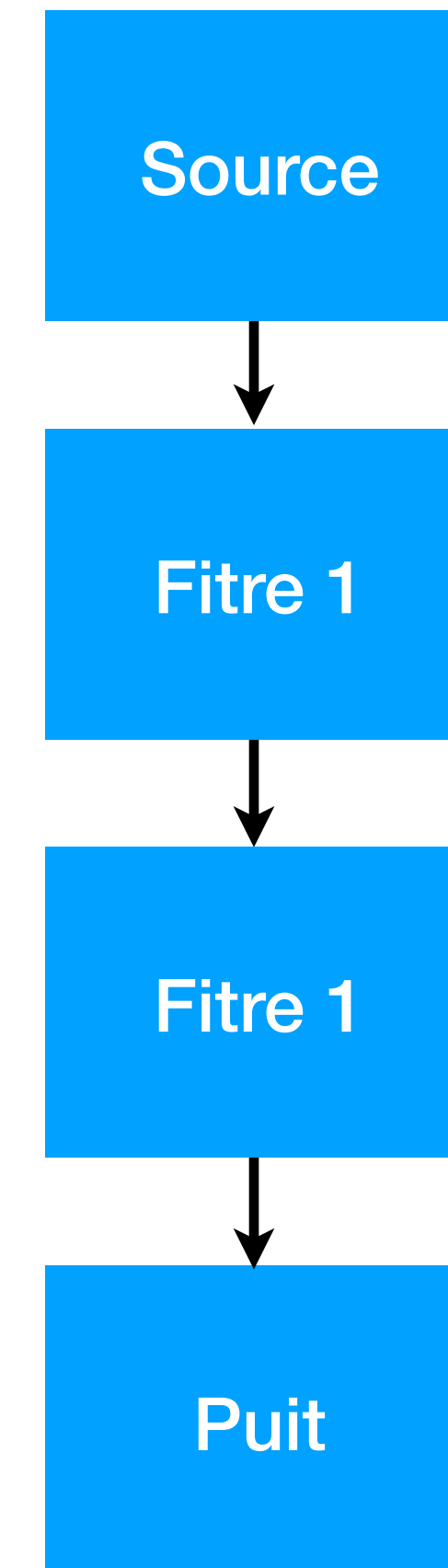
- Inspiré des "pipe" UNIX
- Exemple : Compilateurs



Architecture tuyaux et filtres

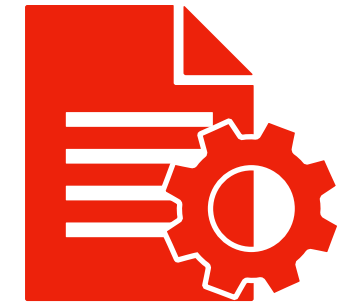
Chaque composant applique une transformation à une entrée

- Inspiré des "pipe" UNIX
- Exemple : Compilateurs



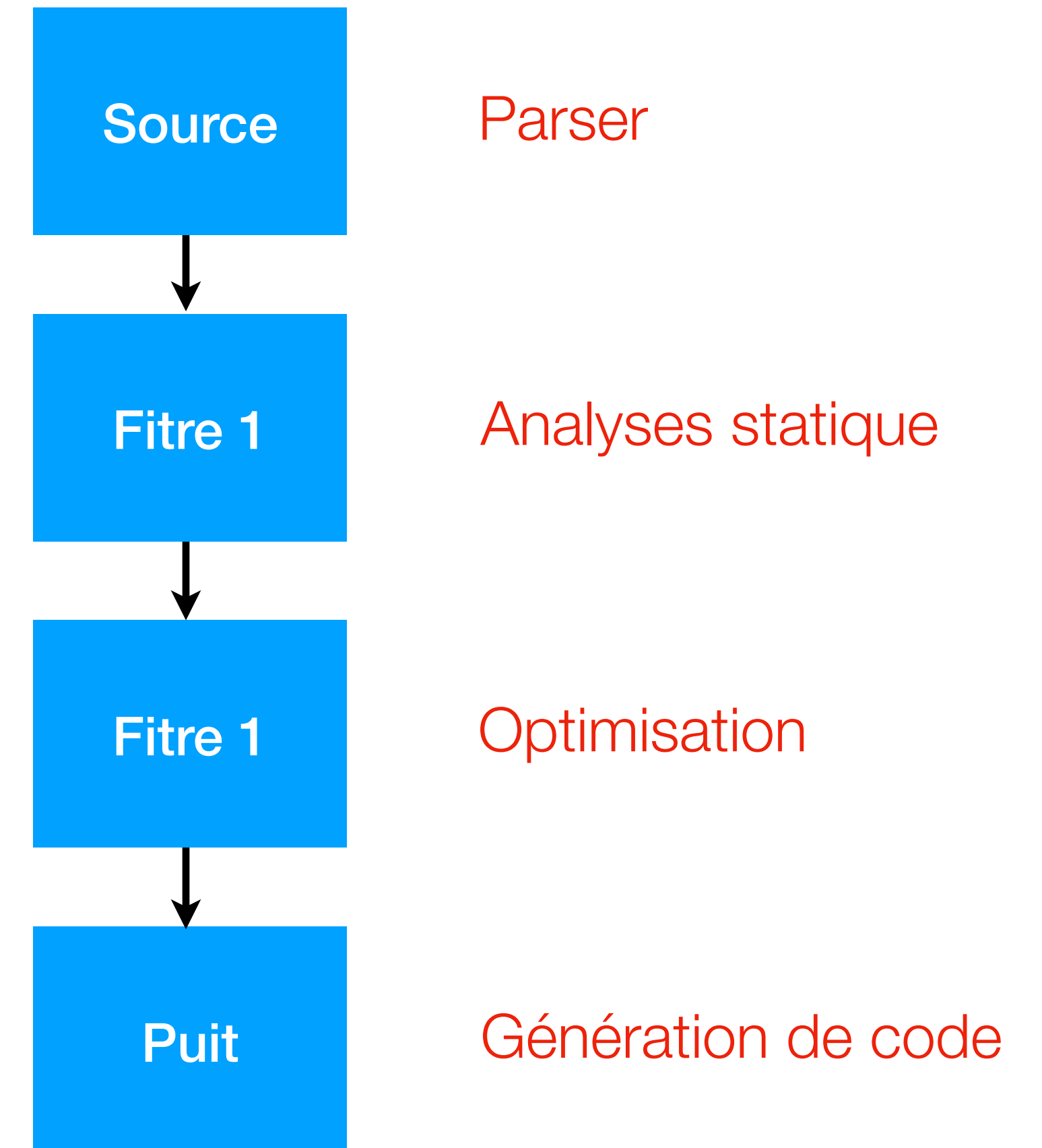
```
cat shakespeare.txt | tr ' ' '\n' | sort | uniq -c | sort -r | head
```

Architecture tuyaux et filtres



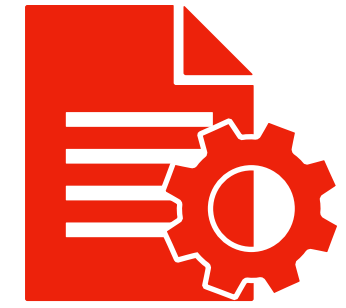
Chaque composant applique une transformation à une entrée

- Inspiré des "pipe" UNIX
- Exemple : Compilateurs



```
cat shakespeare.txt | tr ' ' '\n' | sort | uniq -c | sort -r | head
```

Architecture tuyaux et filtres

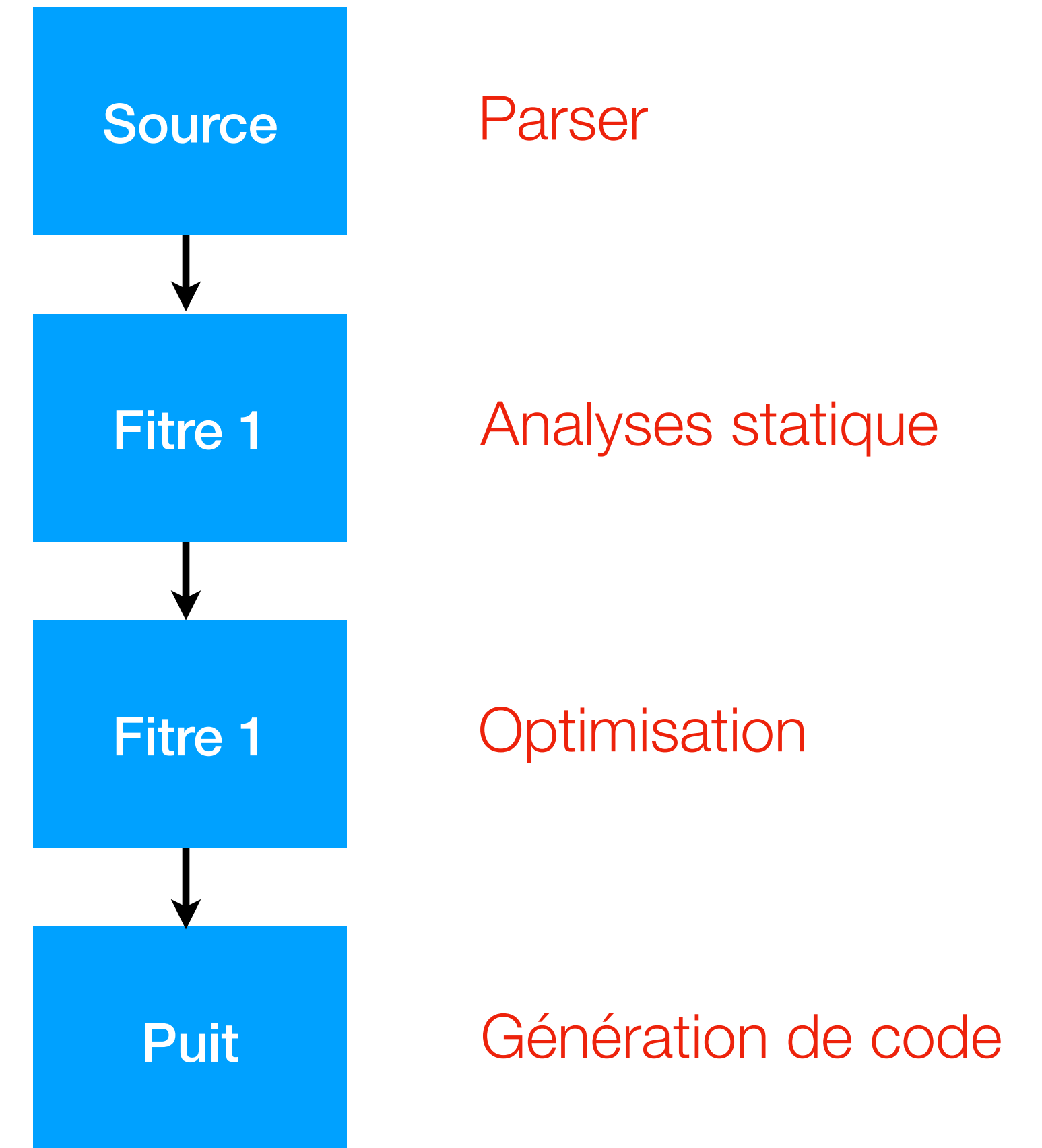


Chaque composant applique une transformation à une entrée

- Inspiré des "pipe" UNIX
- Exemple : Compilateurs

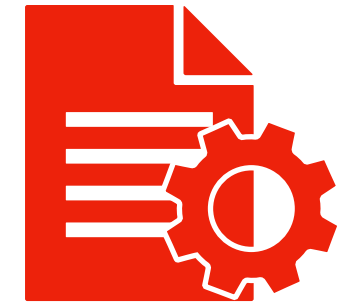
Avantages

- Modularité
- Facile à maintenir et à faire évoluer
- Architecture distribuée (tri topologique)



```
cat shakespeare.txt | tr ' ' '\n' | sort | uniq -c | sort -r | head
```

Architecture tuyaux et filtres



Chaque composant applique une transformation à une entrée

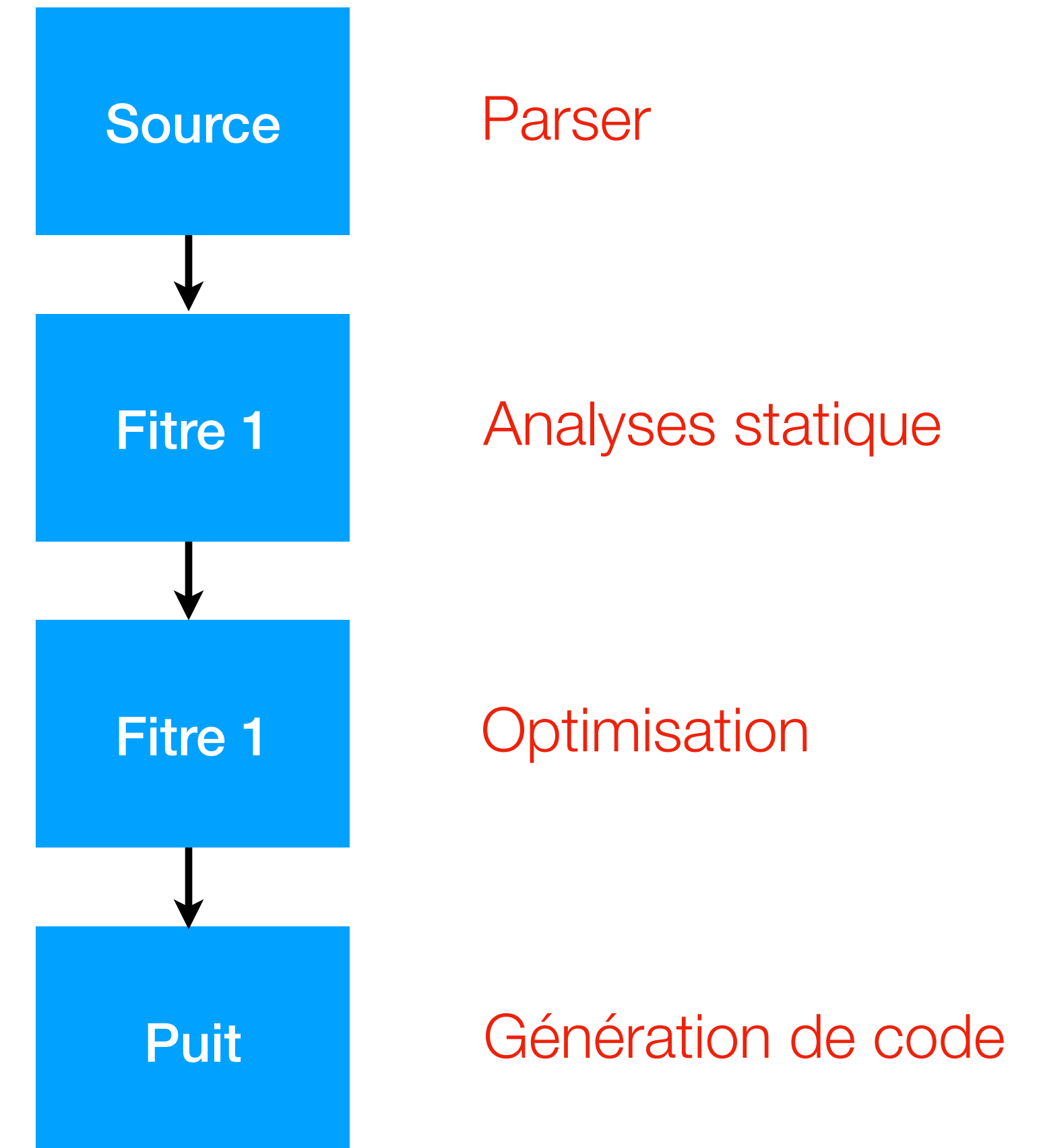
- Inspiré des "pipe" UNIX
- Exemple : Compilateurs

Avantages

- Modularité
- Facile à maintenir et à faire évoluer
- Architecture distribuée (tri topologique)

Inconvénients

- Format unique figé pour tous les composants
- Performances dégradées (read/write)
- Composants incompatibles



```
cat shakespeare.txt | tr ' ' '\n' | sort | uniq -c | sort -r | head
```


Implémentation



Implémentation

Spécifications

- Prototypage, implémentation
- Documentation
- Pre-condition, post-condition, invariants
- Exemples d'utilisation



Implémentation

Spécifications

- Prototypage, implémentation
- Documentation
- Pre-condition, post-condition, invariants
- Exemples d'utilisation

Test

- Utilisation de jeu de test
- Gestion des bugs
- Couverture de test



Implémentation

Spécifications

- Prototypage, implémentation
- Documentation
- Pre-condition, post-condition, invariants
- Exemples d'utilisation

Test

- Utilisation de jeu de test
- Gestion des bugs
- Couverture de test

Modularité

- Utilisation d'API (Application Programming Interface)
- Créer des modules et les documenter



Validation

Plusieurs techniques possible

- Test (à tous les niveau)
- Analyses statiques, e.g., typage
- *Model checking*, e.g., z3
- Preuve, e.g., méthode B, assistants de preuve
- Inspection de code

E. Dijkstra, 1969

Testing shows the presence, not the absence of bugs.

- Validation : ce que le système est supposé faire
- Défaut : réaction aux erreurs

Validation

Plusieurs techniques possible

- Test (à tous les niveau)
- Analyses statiques, e.g., typage
- *Model checking*, e.g., z3
- Preuve, e.g., méthode B, assistants de preuve
- Inspection de code

E. Dijkstra, 1969

Testing shows the presence, not the absence of bugs.

- Validation : ce que le système est supposé faire
- Défaut : réaction aux erreurs

```
import pytest

def add_one(x: int) → int:
    """
    Returns its integer argument plus one
    """
    assert isinstance(x, int)
    return x + 1

def test_ok():
    assert add_one(0) == 1
    assert add_one(2) == 3
    assert add_one(-1) == 0

def test_ko():
    with pytest.raises(AssertionError):
        add_one(1.0)
    with pytest.raises(AssertionError):
        add_one({})
    with pytest.raises(AssertionError):
        add_one("foo")
```

```
===== test session starts =====

b.py::test_ok PASSED [ 50%]
b.py::test_ko PASSED [100%]

===== 2 passed in 0.02s =====
```

Validation : hierarchie de test

Validation : hierarchie de test

Test d'acceptation

- Est-ce que le système complet fonctionne comme attendu ?
- Un test par fonctionnalité
- Exemple : *L'utilisateur doit pouvoir montrer son pass sanitaire*

Validation : hierarchie de test

Test d'acceptation

- Est-ce que le système complet fonctionne comme attendu ?
- Un test par fonctionnalité
- Exemple : *L'utilisateur doit pouvoir montrer son pass sanitaire*

Test d'intégration

- Est-ce que le code interagit correctement avec d'autres composants ?
- Intégration avec des composants qu'on ne peut pas modifier.
- Exemple : *La fonction serialize doit renvoyer le format accepté par deserialize*

Validation : hierarchie de test

Test d'acceptation

- Est-ce que le système complet fonctionne comme attendu ?
- Un test par fonctionnalité
- Exemple : *L'utilisateur doit pouvoir montrer son pass sanitaire*

Test d'intégration

- Est-ce que le code interagit correctement avec d'autres composants ?
- Intégration avec des composants qu'on ne peut pas modifier.
- Exemple : *La fonction serialize doit renvoyer le format accepté par deserialize*

Test unitaire

- Est-ce que chaque composant fait ce qu'il est censé faire ?
- Exemple : *`length([]) = 0 ; length([1,2, 3]) = 3 ; length({}) assert False`*

Validation : hierarchie de test

Test d'acceptation

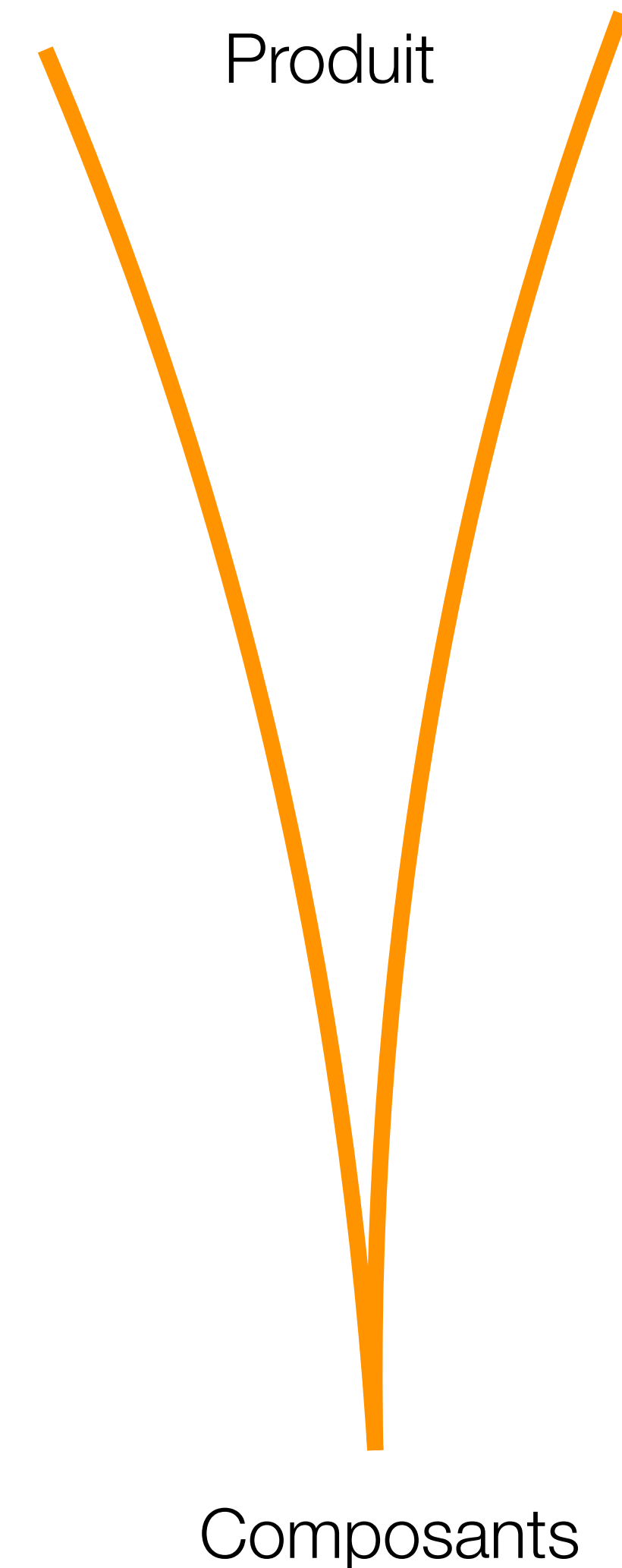
- Est-ce que le système complet fonctionne comme attendu ?
- Un test par fonctionnalité
- Exemple : *L'utilisateur doit pouvoir montrer son pass sanitaire*

Test d'intégration

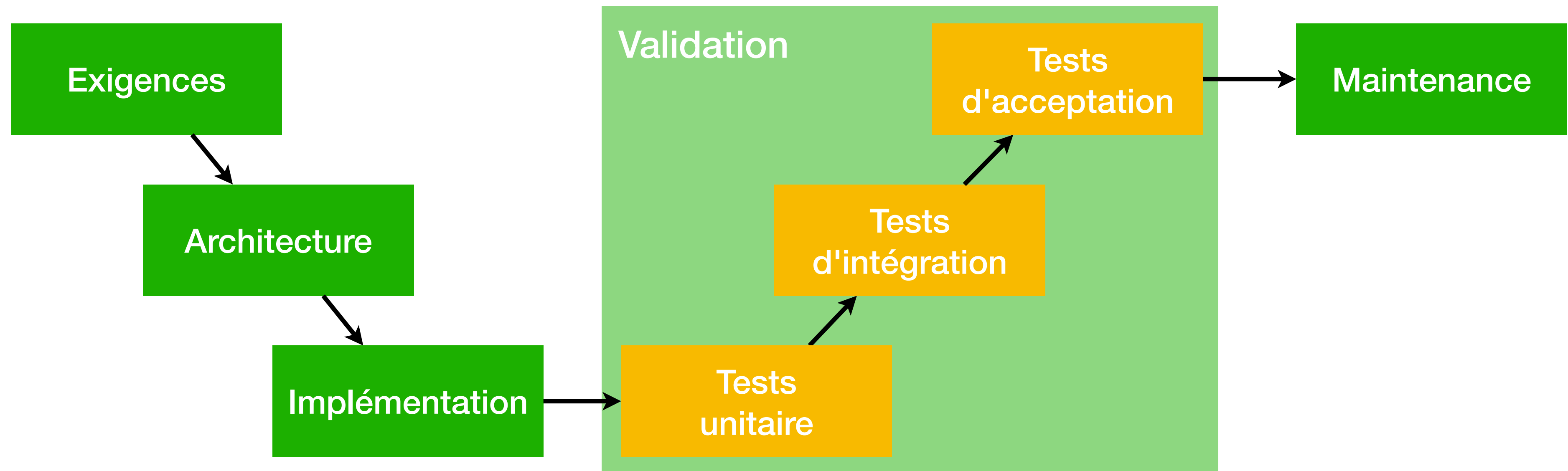
- Est-ce que le code interagit correctement avec d'autres composants ?
- Intégration avec des composants qu'on ne peut pas modifier.
- Exemple : *La fonction serialize doit renvoyer le format accepté par deserialize*

Test unitaire

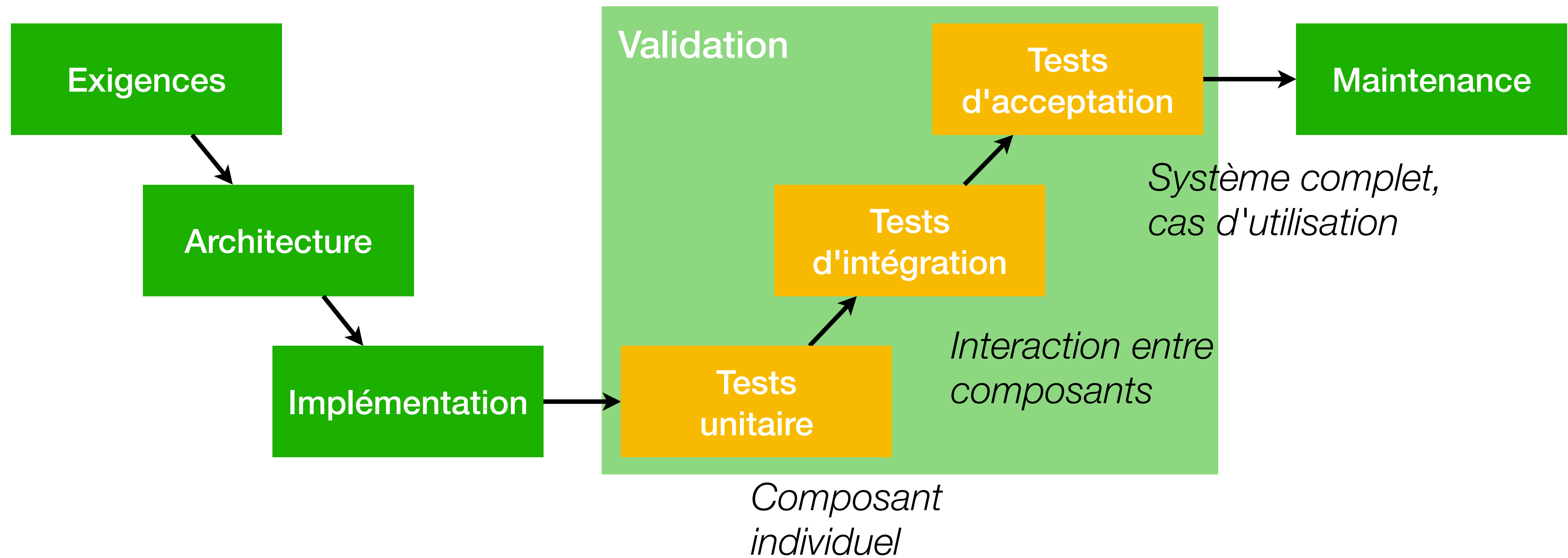
- Est-ce que chaque composant fait ce qu'il est censé faire ?
- Exemple : *length([]) = 0 ; length([1,2, 3]) = 3 ; length({}) assert False*



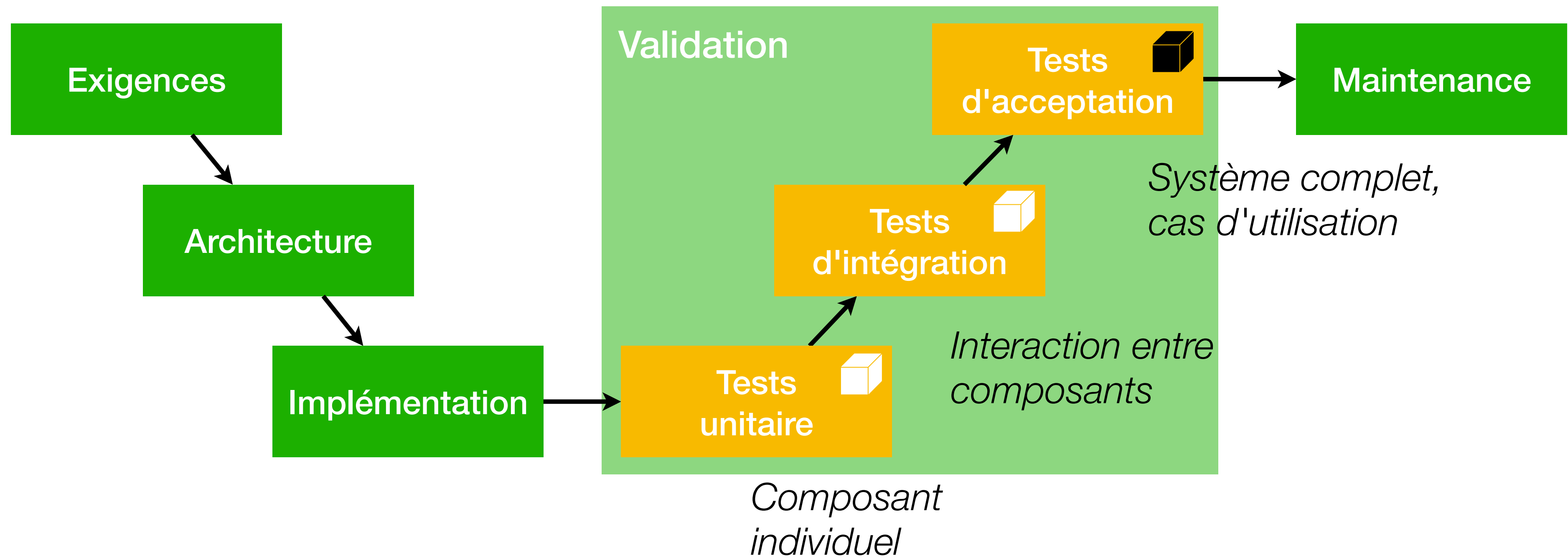
Validation : test



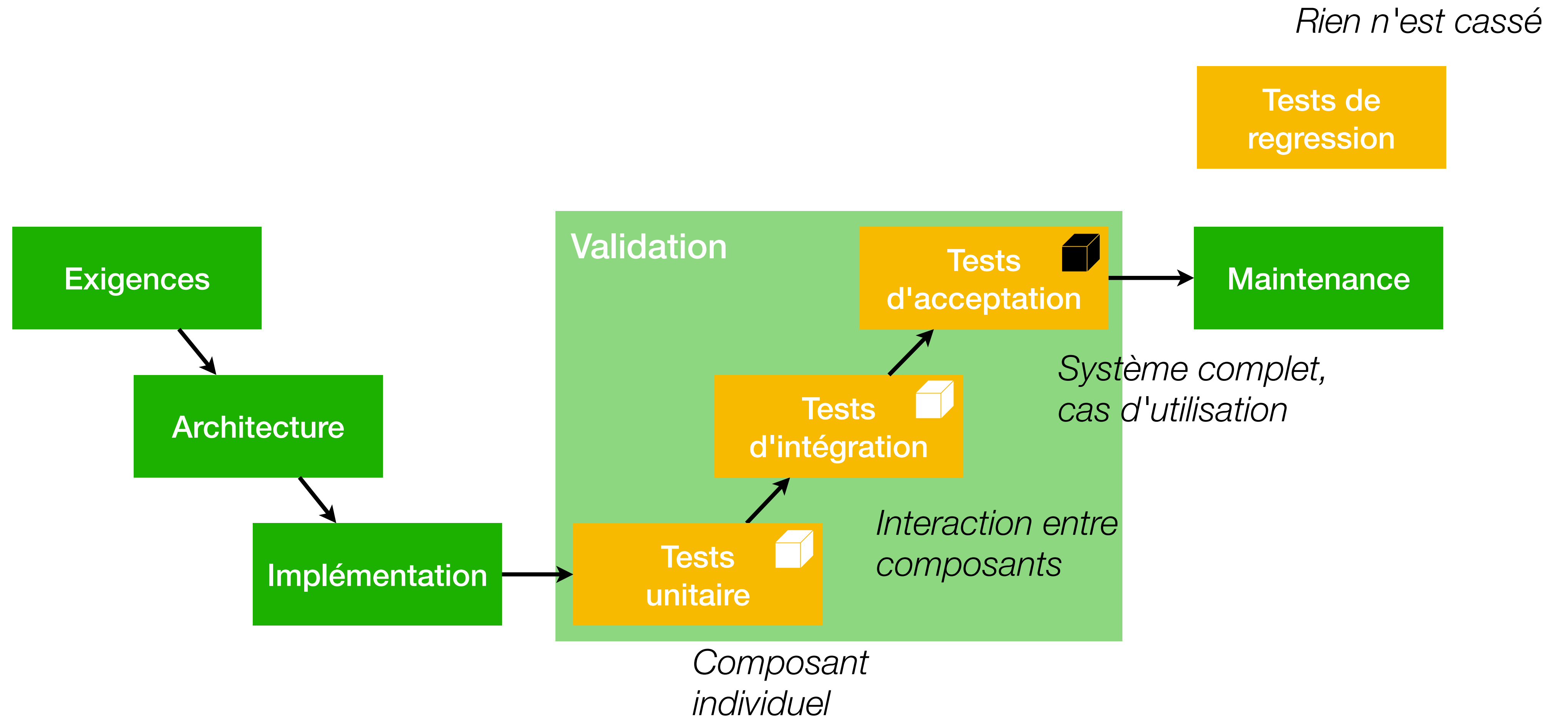
Validation : test



Validation : test



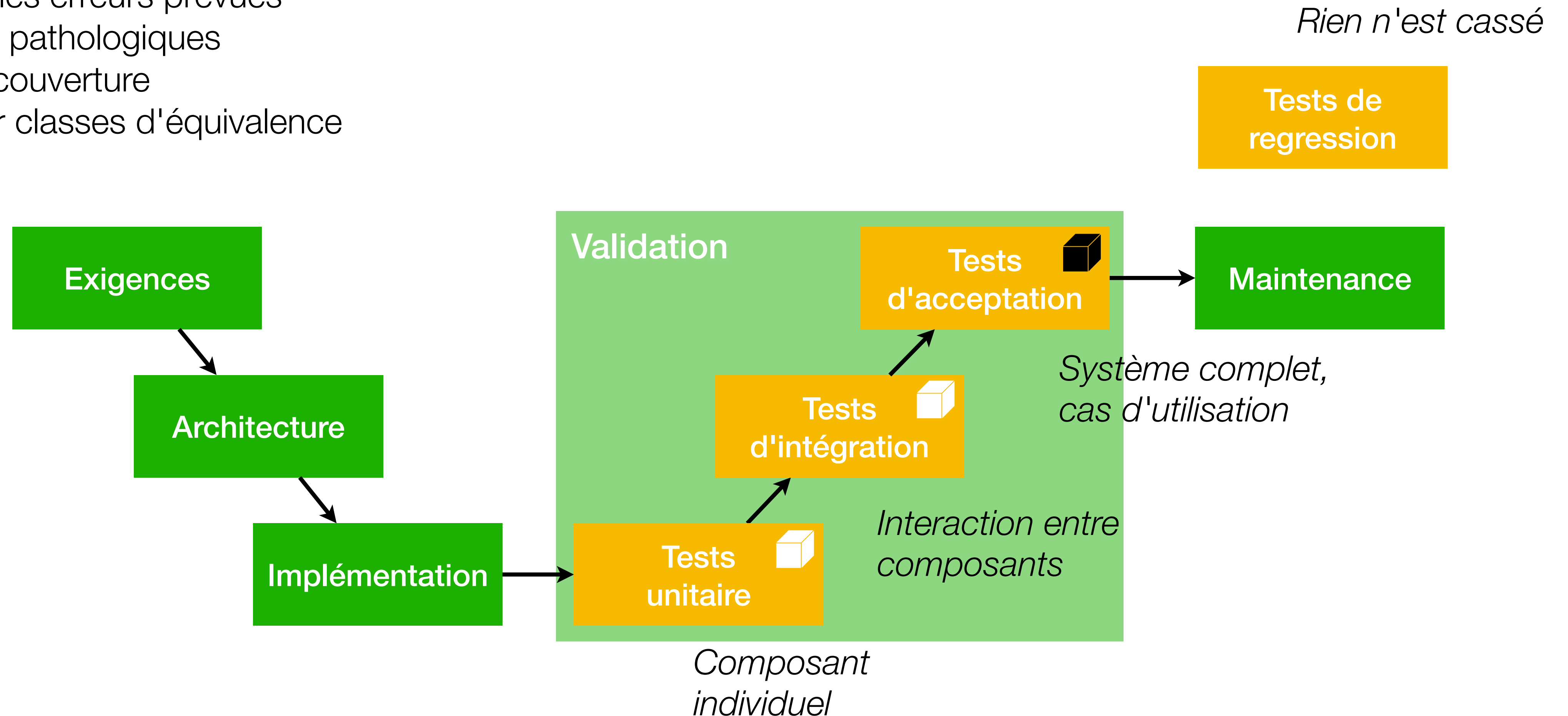
Validation : test




Validation : test

Recommandations

- Forcer les bugs (entrées invalides, débordement)
- Tester toutes les erreurs prévues
- Tester les cas pathologiques
- Maximiser la couverture
- Reasonner par classes d'équivalence



Évolution

 **We found potential security vulnerabilities in your dependencies.**

Some of the dependencies defined in your `package-lock.json` have known security vulnerabilities and should be updated.

[Dismiss](#)

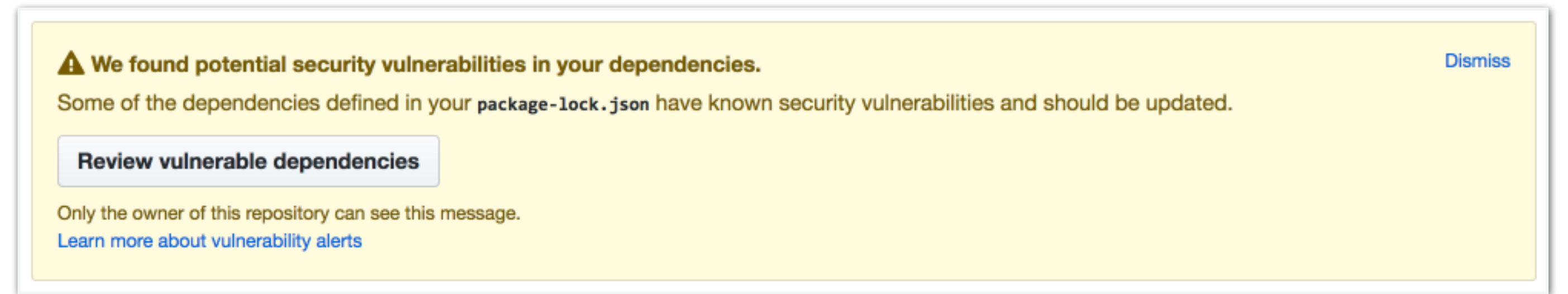
[Review vulnerable dependencies](#)

Only the owner of this repository can see this message.
[Learn more about vulnerability alerts](#)

Évolution

Inévitable...

- Nouvelles exigences, nouveaux usages
- Découverte de vulnérabilités
- Bugs !
- Évolution du matériel



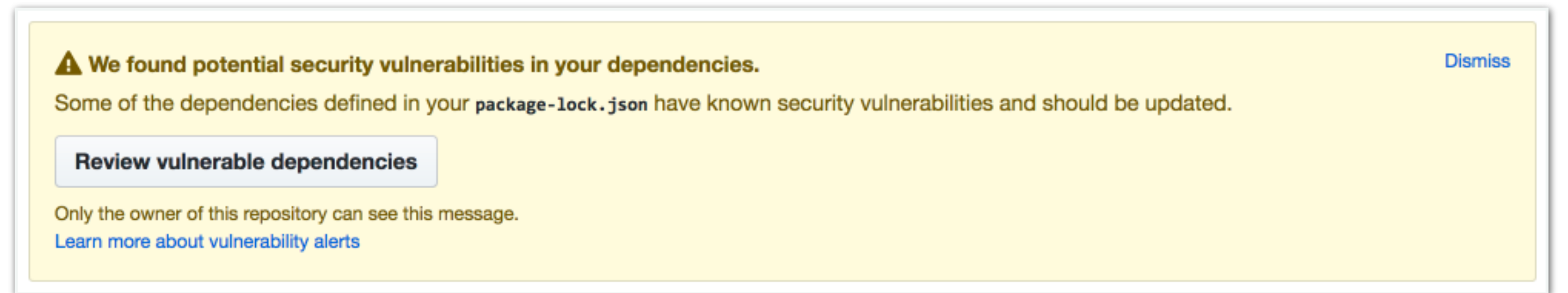
Évolution

Inévitable...

- Nouvelles exigences, nouveaux usages
- Découverte de vulnérabilités
- Bugs !
- Évolution du matériel

Cycle classique

- Évolution : le système change et s'adapte
- Service : simple maintenance
- Retraite : le logiciel n'est plus maintenu



Évolution

Inévitable...

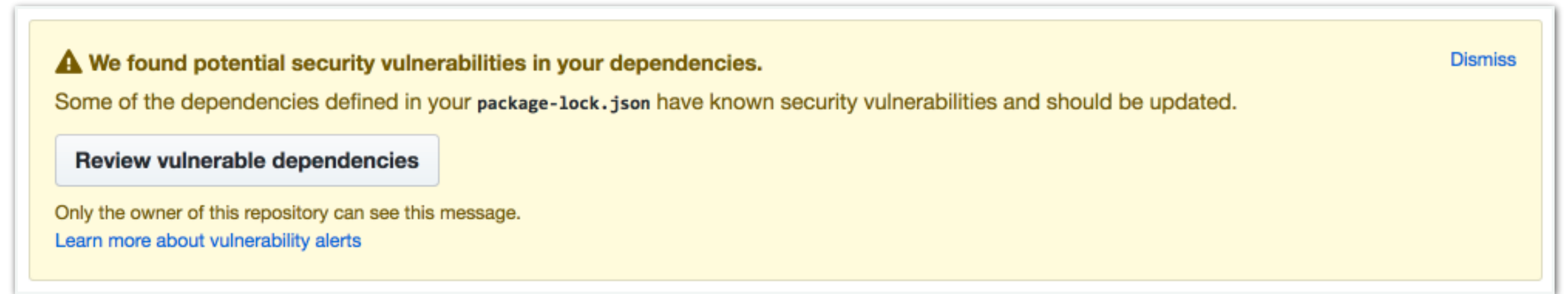
- Nouvelles exigences, nouveaux usages
- Découverte de vulnérabilités
- Bugs !
- Évolution du matériel

Cycle classique

- Évolution : le système change et s'adapte
- Service : simple maintenance
- Retraite : le logiciel n'est plus maintenu

Problèmes

- Changement d'équipe
- Changement de méthode
- Héritage de code
- Typiquement plus coûteux que la conception



Évolution

Inévitable...

- Nouvelles exigences, nouveaux usages
- Découverte de vulnérabilités
- Bugs !
- Évolution du matériel

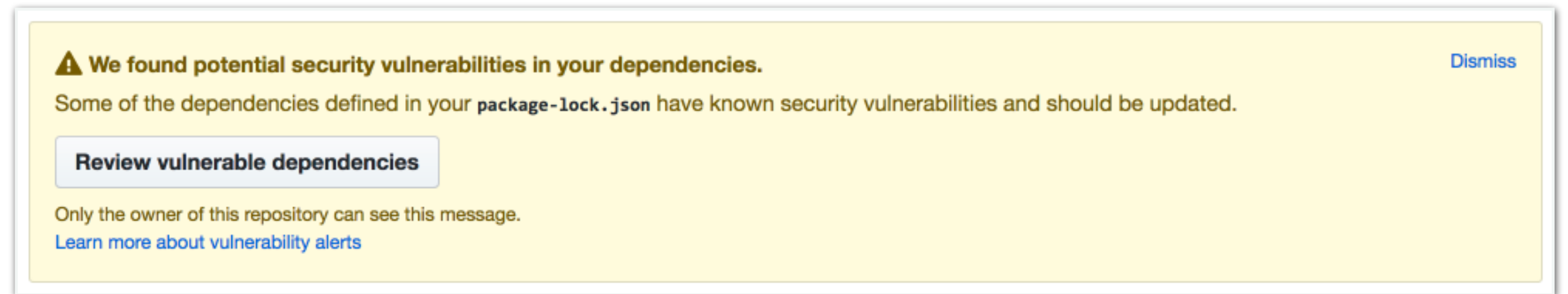
Cycle classique

- Évolution : le système change et s'adapte
- Service : simple maintenance
- Retraite : le logiciel n'est plus maintenu

Problèmes

- Changement d'équipe
- Changement de méthode
- Héritage de code
- Typiquement plus coûteux que la conception

Anticiper dès le départ !



Évolution

Inévitable...

- Nouvelles exigences, nouveaux usages
- Découverte de vulnérabilités
- Bugs !
- Évolution du matériel

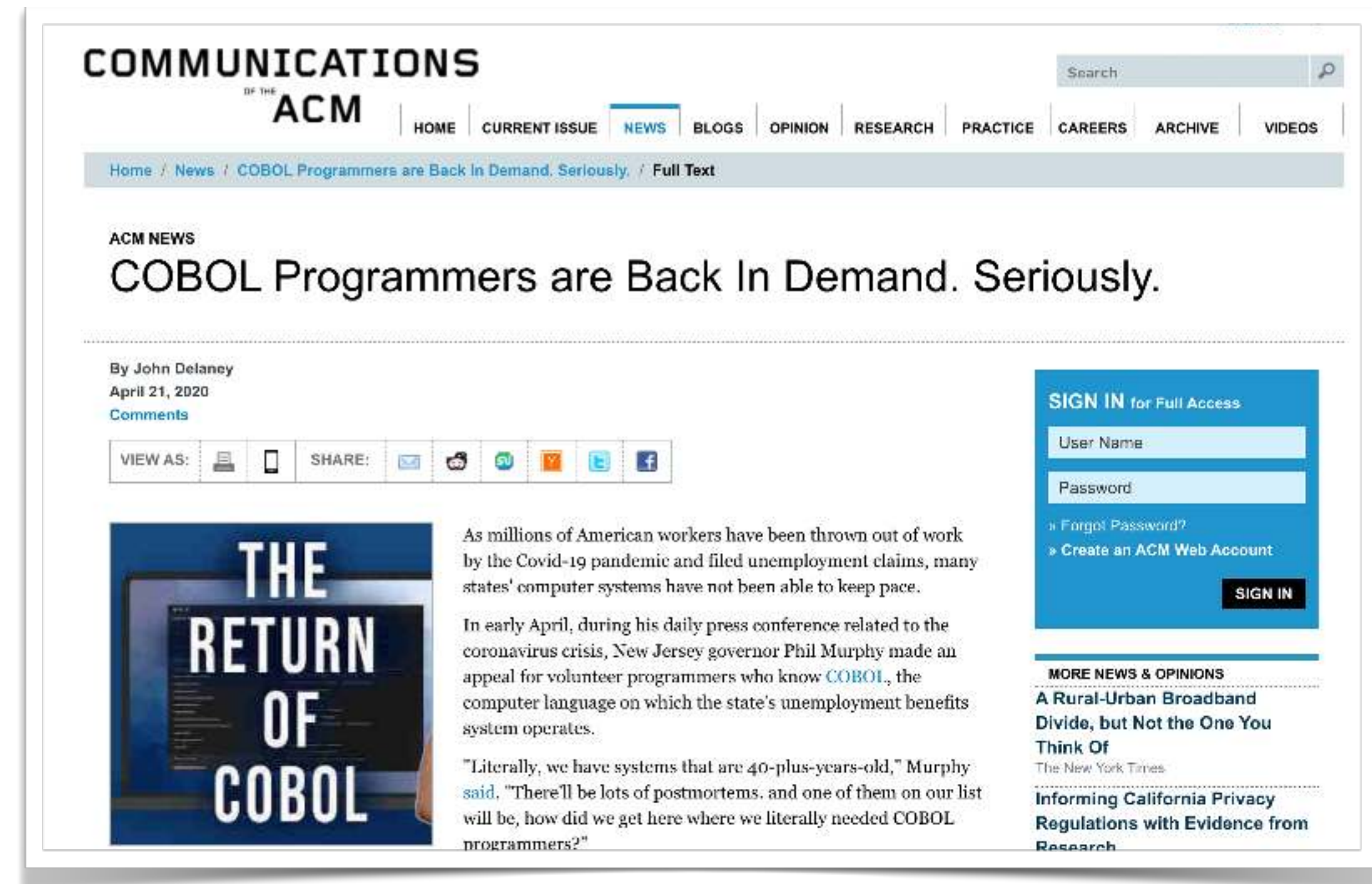
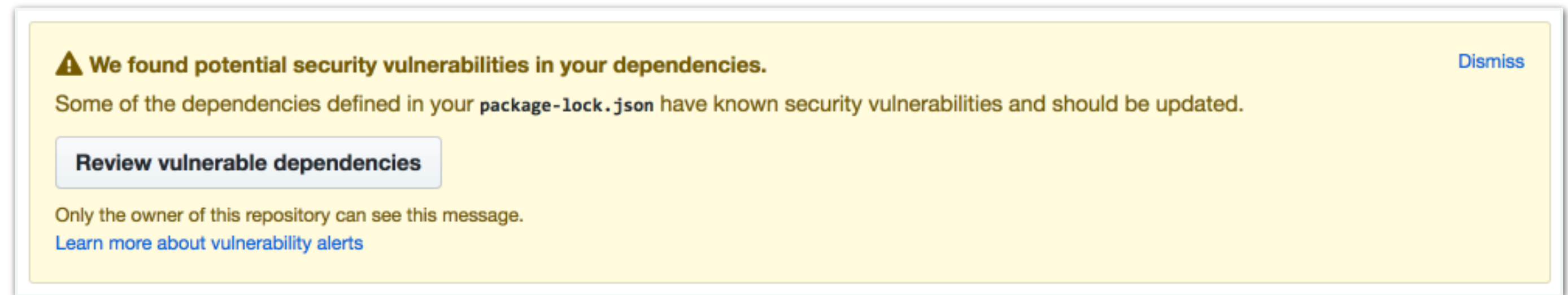
Cycle classique

- Évolution : le système change et s'adapte
- Service : simple maintenance
- Retraite : le logiciel n'est plus maintenu

Problèmes

- Changement d'équipe
- Changement de méthode
- Héritage de code
- Typiquement plus coûteux que la conception

Anticiper dès le départ !



UML

Génie logiciel

UML : Unified Modeling Language

UML : Unified Modeling Language

Notation pour représenter, spécifier, concevoir, documenter, communiquer

- Apparaît en 1995
- Standardisé par l'OMG (*Object Management Group*)
- Depuis 2005 : ISO/IEC 19505

UML : Unified Modeling Language

Notation pour représenter, spécifier, concevoir, documenter, communiquer

- Apparaît en 1995
- Standardisé par l'OMG (*Object Management Group*)
- Depuis 2005 : ISO/IEC 19505

Plusieurs types de diagrammes

- Diagrammes de structure : classes, structure, composants, ...
- Diagramme de comportement : cas d'utilisation, séquence, activité, ...

UML : Unified Modeling Language

Notation pour représenter, spécifier, concevoir, documenter, communiquer

- Apparaît en 1995
- Standardisé par l'OMG (*Object Management Group*)
- Depuis 2005 : ISO/IEC 19505

Plusieurs types de diagrammes

- Diagrammes de structure : classes, structure, composants, ...
- Diagramme de comportement : cas d'utilisation, séquence, activité, ...

Exigences

Cas d'utilisation
Séquence

Architecture

Classes
Composants

Implémentation

Activités

Validation

Séquence

UML : Unified Modeling Language

Notation pour représenter, spécifier, concevoir, documenter, communiquer

- Apparaît en 1995
- Standardisé par l'OMG (*Object Management Group*)
- Depuis 2005 : ISO/IEC 19505

Plusieurs types de diagrammes

- Diagrammes de structure : classes, structure, composants, ...
- Diagramme de comportement : cas d'utilisation, séquence, activité, ...

Exigences

Cas d'utilisation
Séquence

Architecture

Classes
Composants

Implémentation

Activités

Validation

Séquence

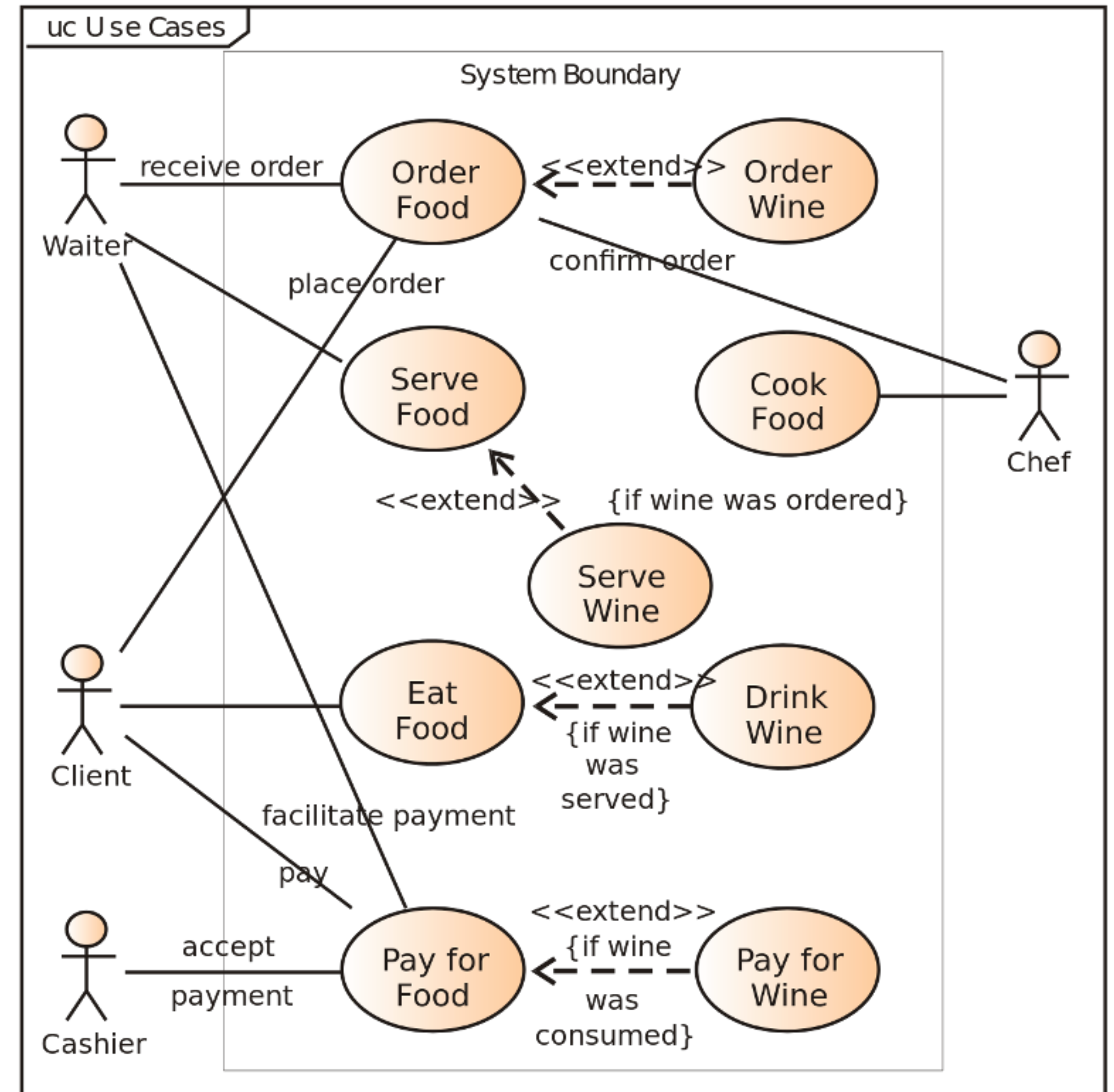
Limites

- Pas un langage de programmation
- Pas un langage formel

UML : diagramme de cas d'utilisation

Décrit interaction avec les utilisateurs

- Personnage : utilisateurs du système
- Ellipse : fonctionnalité
- Lien : acteur / utilisation



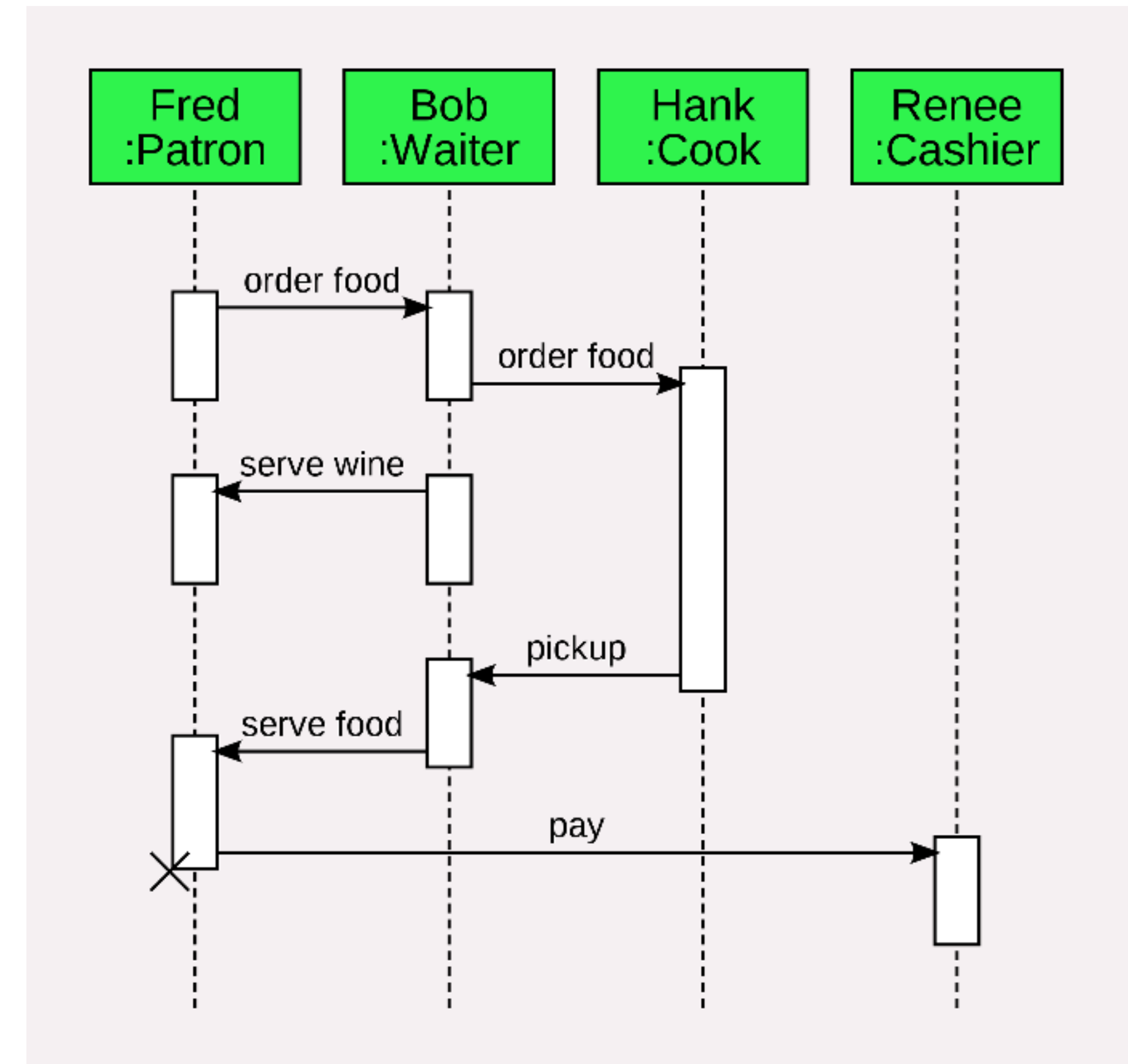
UML : diagramme de séquence

Décrit interactions avec le système

- Ligne de vie : participants
- Communication par message
- Suit un ordre chronologique

Utilisation

- Formaliser les exigences
- Cas de test



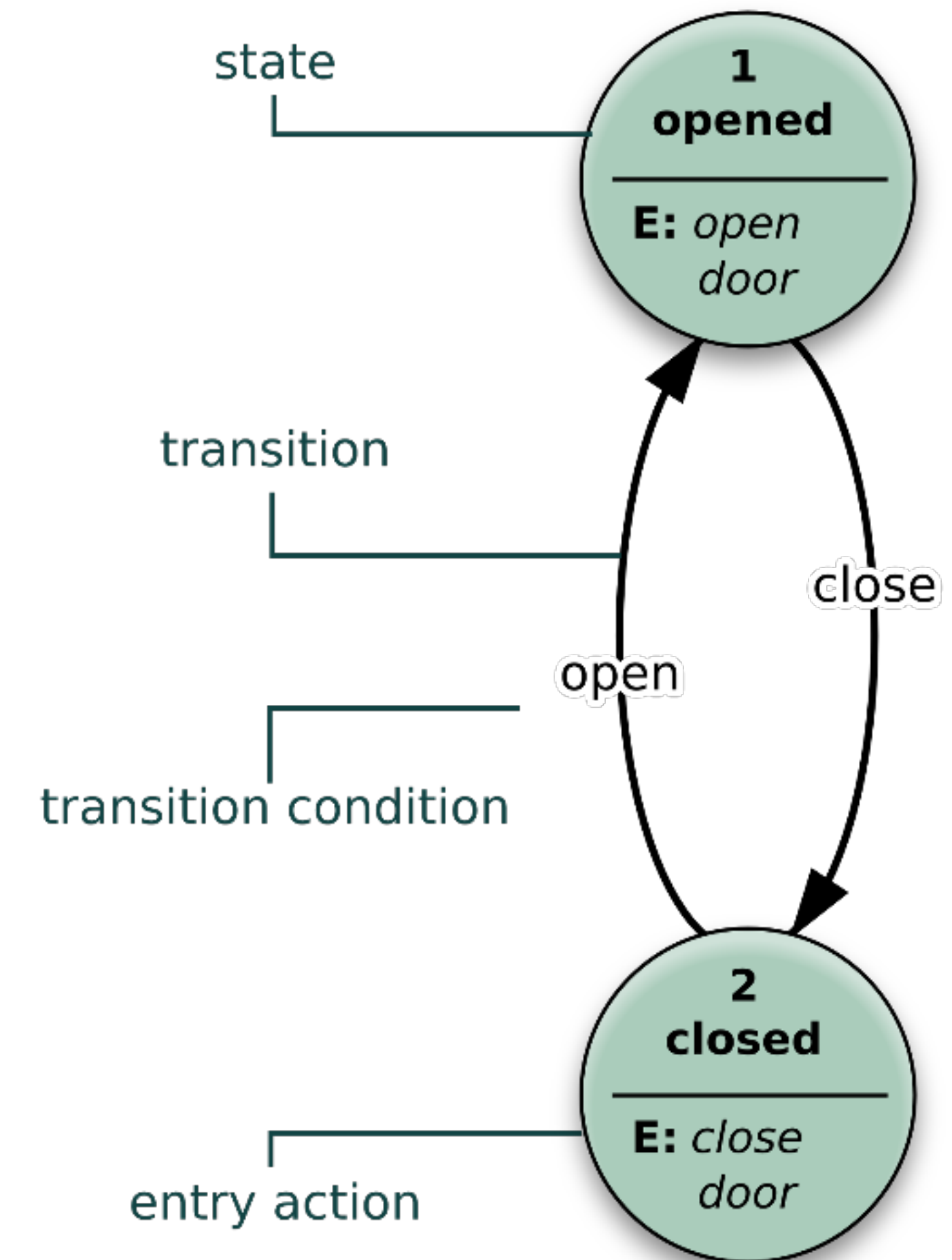
UML: Diagramme états-transitions

Décrit le comportement d'un logiciel

- État: satisfait un ensemble de conditions
- Transition: changement d'état

Utilisation

- Vue synthétique dynamique
- Capture un ensemble de scénarios



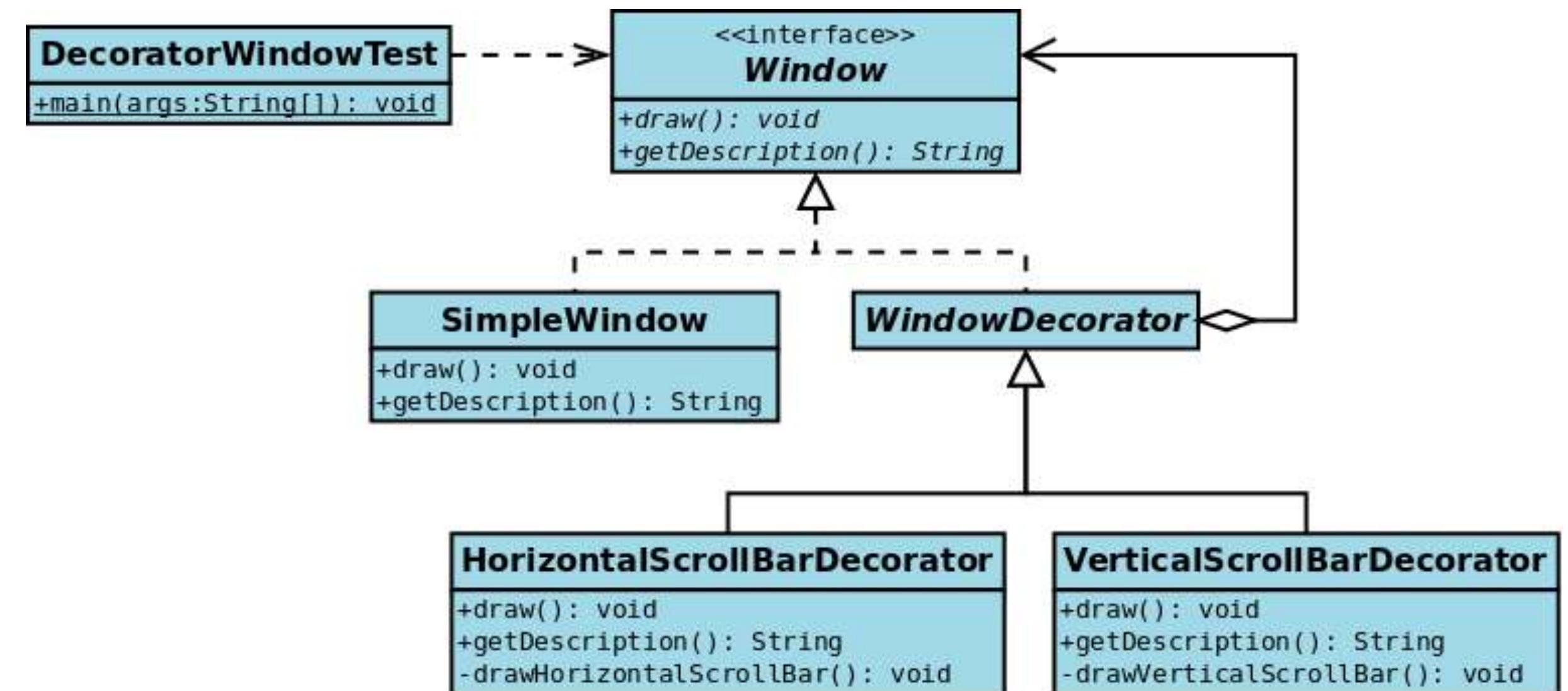
UML : diagramme de classe

Programmation orientée objet

- Boîtes: Classe, attributs, méthodes
- Lien : héritage, association, composition, ...

Patron de conceptions

- GoF 1994 *Design Patterns, Elements of Reusable Object Oriented Software*
- Solution abstraite qui peut être réutilisée
- Analyse de code



Git

Génie Logiciel

Git

Git

Git : Global Information Tracker (ou *the stupid content tracker*)

- 2005 : Initialement développé par L. Torvald pour le noyau Linux
- Conçu comme un système de fichier versionné

Git

Git : Global Information Tracker (ou *the stupid content tracker*)

- 2005 : Initialement développé par L. Torvald pour le noyau Linux
- Conçu comme un système de fichier versionné

As of January 2020, GitHub reports having over 40 million users and more than 190 million repositories (including at least 28 million public repositories).

Git

Git : Global Information Tracker (ou *the stupid content tracker*)

- 2005 : Initialement développé par L. Torvald pour le noyau Linux
- Conçu comme un système de fichier versionné

As of January 2020, GitHub reports having over 40 million users and more than 190 million repositories (including at least 28 million public repositories).

Trained on billions of lines of public code, GitHub Copilot puts the knowledge you need at your fingertips, saving you time and helping you stay focused.

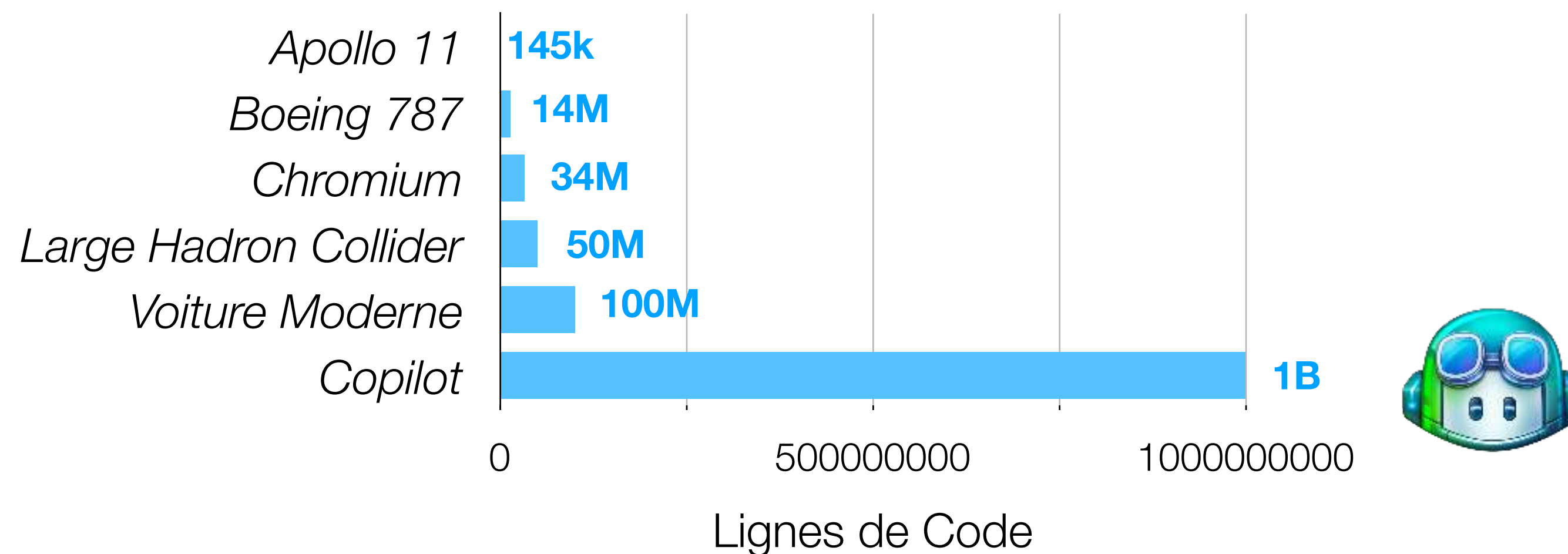
Git

Git : Global Information Tracker (ou *the stupid content tracker*)

- 2005 : Initialement développé par L. Torvald pour le noyau Linux
- Conçu comme un système de fichier versionné

As of January 2020, GitHub reports having over 40 million users and more than 190 million repositories (including at least 28 million public repositories).

Trained on billions of lines of public code, GitHub Copilot puts the knowledge you need at your fingertips, saving you time and helping you stay focused.



Git : rappel

```
git clone myrepo .  
git add foo  
git commit -m "initial commit"  
git push
```

```
git checkout -b dev  
git add bar  
git commit -m "-- "  
git checkout main  
git merge dev
```

```
git log  
git status  
git reflog
```

```
git pull  
git pull --rebase  
git rebase -i main
```

```
# Clone un dépôt  
# Ajoute un fichier dans l'index  
# Créé un nouveau commit  
# Pousse les changement sur origin
```

```
# Créé une nouvelle branche
```

```
# Message informatif  
# Revenir sur main  
# Fusionner les branches
```

```
# Affiche l'histoire  
# Affiche les differences index/état courant  
# Affiche toutes les operations
```

```
# Ajoute les modifications du dépôt distant  
# Repositionne la base  
# Fusionne les commits (interactif)
```

MiniGit

```
minigit init  
minigit add foo  
minigit commit -m "Add foo"  
minigit log
```

```
minigit branch bar  
minigit checkout bar  
minigit add bar  
minigit commit -m "Add bar"  
minigit log
```

```
minigit checkout main  
minigit log
```

