Découpler tes composants

Dans de nombreux projets réalisés en classe, tu travailles en équipe sur un système composé de plusieurs éléments : une API, des classes métier, un micro-contrôleur, un programme “bridge” sur PC, ou encore une page web qui affiche des graphiques.

5TTR 6TTR Découverte

Travailler en parallèle grâce au contrat d’échange entre composants

Un blocage revient très souvent dans les phases de travail de la classe:

> “JE NE PEUX PAS AVANCER, J’ATTENDS QUE L’AUTRE AIT FINI.”

Ce cours part de cette situation réelle pour montrer pourquoi **cette idée est fausse**, et comment une bonne définition de l’interface d’échange permet à **plusieurs équipes de travailler en parallèle**, sans dépendre du code des autres.

Objectifs d’apprentissage

À la fin de cette leçon, tu dois être capable de :

- comprendre pourquoi attendre le code d’un autre est une erreur de raisonnement ;
- expliquer le rôle d’une interface d’échange comme **contrat** ;
- travailler avec des **mock data** pour avancer sans dépendances ;
- intégrer des composants développés séparément sans tout casser ;
- mettre un nom professionnel sur ce principe.

Situation réelle vécue par les élèves

Projet typique :

- un micro-contrôleur envoie des mesures par radio ou via un port série ;
- un programme “bridge” (Python ou JavaScript) récupère ces données ;
- une API ou un module les traite ;
- une page web affiche un graphique en temps réel.

Très souvent, on entend :

> “Je ne peux pas coder la page web, je n’ai pas encore les données.” “Je dois attendre que l’autre groupe ait fini le micro-contrôleur.”

👉 **C’est précisément là que se situe l’erreur.**

L’erreur de raisonnement classique

Le problème n’est **pas** que l’autre composant n’est pas terminé. Le vrai problème est que **l’interface d’échange n’est pas clairement définie**.

Beaucoup d’élèves pensent dépendre :

- du code de l’autre,
- de sa base de données,
- de son matériel.

En réalité, tu ne dépends **que de la forme des données** que tu vas recevoir.

L’interface d’échange : un contrat entre composants

Une **interface d’échange** définit **ce qui circule entre deux composants**, pas comment ils sont codés.

> **CONTRAT D’ÉCHANGE** Accord formel entre deux composants sur :
> - le format des données,
> - les champs attendus,

- les types,
- les unités,
- les règles minimales (présence, valeurs possibles).

Si **les deux équipes respectent le contrat**, alors :

- elles peuvent travailler **en parallèle** ;
- elles n'ont **pas besoin du code de l'autre** ;
- l'intégration finale devient simple.

Exemple concret 1 : micro-contrôleur → bridge → web

Contrat d'échange (exemple JSON)

```json
{
  "timestamp": 1710001234,
  "temperature": 22.6,
  "humidity": 48.2
}
```

Ce contrat précise :

- les noms des champs,
- les types (`number`),
- le sens des valeurs (°C, %),
- la structure globale.

À partir de là :

- l'équipe **micro-contrôleur** s'engage à produire ces données ;
- l'équipe **bridge / web** s'engage à consommer ce format.

Travailler sans attendre : les mock data

> **MOCK DATA** Données fictives mais réalistes, qui respectent le contrat d'échange.

Exemple en JavaScript (frontend)

```javascript
const mockData = {
  timestamp: 1710001234,
  temperature: 21.8,
  humidity: 50.1
};

drawGraph(mockData);
```

Exemple en Python (backend ou bridge)

```python
mock_data = {
    "timestamp": 1710001234,
    "temperature": 23.1,
    "humidity": 46.9
}

process_measure(mock_data)
```

👉 Le code est **le même** que celui utilisé plus tard avec les vraies données.

Exemple concret 2 : classes qui collaborent

Deux classes développées par deux élèves différents :

- `SensorReader` (lecture des données)
- `DataAnalyzer` (analyse et statistiques)

Contrat d'échange

```python
{
    "value": float,
    "unit": str,
    "source": str
}
```

Tant que ce dictionnaire est respecté :

- `DataAnalyzer` peut être développé **avant** `SensorReader` ;
- les tests peuvent utiliser des mocks ;
- aucune dépendance directe n'existe entre les deux implémentations.

Intégration finale : le moment de vérité

Quand les deux composants sont terminés :

- on remplace les mock data par les vraies données ;
- si le contrat est respecté, **tout fonctionne sans modification** ;
- sinon, le bug est clair : **le contrat n'a pas été respecté**.

👉 Le problème devient **localisé et compréhensible**, pas diffus.

Impact direct sur les tests

Un système basé sur des contrats permet :

- des tests unitaires isolés ;
- des tests avec données simulées ;
- des tests reproductibles.

Sans contrat :

- les tests dépendent du matériel, du réseau ou d'un autre groupe ;
- les bugs sont difficiles à diagnostiquer ;
- le développement ralentit fortement.

Ce que tu dois retenir

- Tu ne dépends pas du code d'un collègue, tu dépends du **format des données**.
- L'interface d'échange est un **contrat**, pas une implémentation.
- Les mock data permettent de travailler **en parallèle**.
- Si le contrat est respecté, l'intégration est simple.
- Ce principe est utilisé partout dans l'industrie.

Mise en perspective professionnelle

Ce que tu viens de voir porte un nom en architecture logicielle : 👉 **le loose coupling (couplage faible)**.

Tu n’avais pas besoin de connaître le terme pour l’appliquer, mais maintenant tu peux le reconnaître, l’expliquer et l’utiliser correctement.

© 2025 Mr. Lambrechts https://v2.ttrinfo.be 6 | 6