Programmation orientée aspect et API déclarative

Mikaël Francoeur

Responsabilités transversales (cross-cutting concerns)

Problème: certaines choses qu'on veut faire à plusieurs endroits différents.

- Logging
- Tracing
- Métriques
- Sécurité (autorisation)
- Caching
- Audit
- Retries

Programmation orientée aspect (AOP)

Le nom date de 1995-1996, chez Xerox

Premier article scientifique: Kiczales et al. 1997

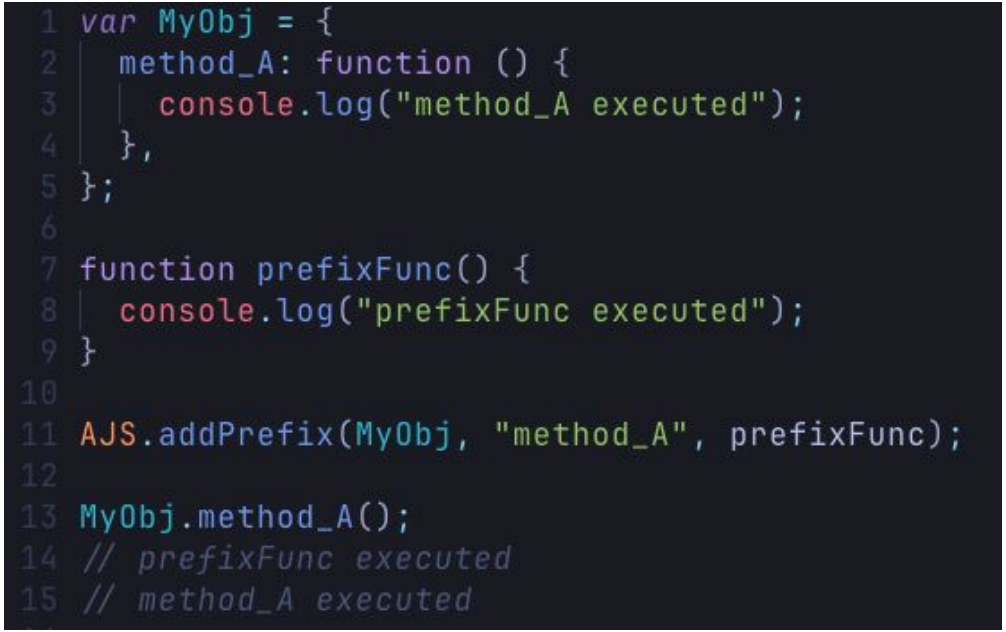
Terminologie (AspectJ, 2001)

- Aspect: une responsabilité transversale
- Advice: un comportement arbitraire à ajouter au programme
- Joinpoint: un endroit ou joindre (join) l'advice
- Pointcut: une sélection de joinpoints

Le problème

- Système patrimonial
- On veut éviter de toucher au code existant
- Besoin: auditer plusieurs dizaines de méthodes
    - La logique est toujours la même
        - envoyer un ou des ids d'utilisateurs, et
        - le type de l'audit (CREATE, DELETE)
    - Le ou les identifiants à auditer sont toujours dans les arguments, mais peuvent être imbriqués

AspectJS (TypeScript)



```
var MyObj = {
  method_A: function () {
    console.log("method_A executed");
  },
};

function prefixFunc() {
  console.log("prefixFunc executed");
}

AJS.addPrefix(MyObj, "method_A", prefixFunc);

MyObj.method_A();
// prefixFunc executed
// method_A executed
```

aspectlib (Python)

```python
@aspectlib.Aspect
def strip_return_value(*args, **kwargs):
    result = yield aspectlib.Proceed
    yield aspectlib.Return(result.strip())

@strip_return_value
def read(name):
    return open(name).read()

```

Avantages:

- Peu invasif.
- La logique d'affaires n'est pas polluée.
- Les cross-cutting concerns sont quand même visibles, mais en dehors de la méthode.
- Facile à réutiliser et à adapter (lib).
- Peut être testé en isolation (Spring)

Désavantages:

- Plus compliqué à écrire initialement qu'un décorateur-GOF
- Peut donner des comportements "magiques" et peu observables
- Plus difficile à déboguer

Questions?