



06

Chapitre

Conclusion sur les patrons de conception

2I1AC3 : Génie logiciel et Patrons de conception

Régis Clouard, ENSICAEN - GREYC

« Les patrons de conception vous aident
à apprendre des succès des autres
plutôt que de vos propres échecs. »

Mark Johnson

Patrons de conception

- Un patron de conception (*design pattern*) est une solution éprouvée à un problème récurrent dans la conception orientée objet de logiciels
 - Pour les utiliser, il faut savoir identifier le problème qu'ils résolvent
- Les patrons sont décrits au niveau conception et sont donc indépendants des langages de programmation

Patrons de conception de la bande des quatre (GOF)

- À leur création, on croyait que ce serait les premiers d'une longue série, mais il s'est avéré que ce sont pratiquement les seuls généraux...
 - 23 patrons (19 sont considérés dans ce cours)

Création	Structure	Comportement
Fabrique simple	Adaptateur	Chaîne de responsabilités
Fabrique abstraite	Composite	Commande
Monteur	Décorateur	État
Prototype	Façade	Interpréteur
Singleton	Poids mouche	Itérateur
	Pont	Médiateur
	Procuration	Memento
		Observateur
		Patron de méthode
		Stratégie
		Visiteur

- À connaître par cœur : ils forment le socle culturel du génie logiciel

Objectif du cours

- L'objectif du cours est surtout centré sur :
 - Sensibilisation à la qualité d'une conception : **cohésion / couplage**
 - ▶ **Cohésion** : Caractérise l'étendue de la responsabilité d'un module
 - ▶ **Couplage** : Caractérise la force d'interaction entre les modules
 - Appropriation des **principes** et des **règles** de conception avancés
- A partir de ces principes et de ces règles, on peut retrouver la plupart des patrons

1. Principes de conception

5

- Votre code est STUPID rendez-le SOLID

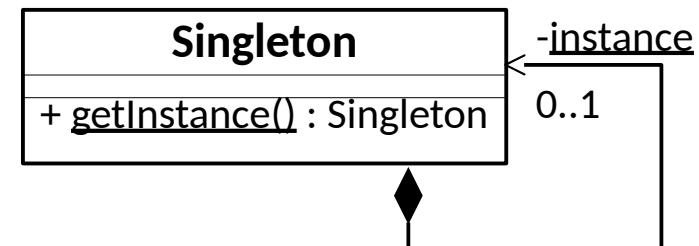
S singleton

T
U
P
I
D

Le patron de conception Singleton

- Garantir qu'une classe ne possède **jamais** plus d'une seule instance
- Principe de base :
 - Dérouter le constructeur et passer par une méthode statique

```
public final class Singleton {  
    private static Singleton _instance; // 1  
    private Singleton() { } // 2  
    public static synchronized Singleton getInstance() { // 3  
        if (_instance == null) {  
            _instance = new Singleton();  
        }  
        return _instance;  
    }  
}
```



- Création d'une instance

```
Singleton s = new Singleton();
```

```
Singleton s = Singleton.getInstance();
```

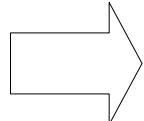
L'anti-patron de conception Singleton

- Un singleton est comparable à une variable globale avec les mêmes défauts :
 - Instance globale au programme : code rigide
 - Utilisation par effets de bord : effets non circonscrits et donc incontrôlables
 - Concurrence : non partageable
 - Intestable : impossible de doubler l'instance (avec des Mock)
- Son utilisation doit être incontournable, une réponse à un problème critique, tel que :
 - La tour de contrôle d'un aéroport.
 - La liste d'attente d'une imprimante 3D en réseau. Si la liste n'est pas partagée, il peut y avoir la fabrication d'objets bizarres.
- Exemple typique de l'anti-patron Singleton : base de données

Principes de conception

- Votre code est STUPID rendez-le SOLID

Singleton
Tight coupling
Untestability
Premature optimization
Indescriptive naming
Duplication



Single Responsibility
Open-Closed
Liskov Substitution
Interface Segregation
Dependency Inversion

2. Règles de conception

Règle 1. Réduire l'**accessibilité** des membres de classe

Règle 2. Encapsuler ce qui **varie** dans une hiérarchie spécifique

Règle 3. Programmer pour une **interface** et non pour une implémentation

Règle 4. Privilégier la **composition** à l'héritage

2. Règles de conception

10

Règle 1. Réduire l'accessibilité des membres de classe

- Ne mettez les accesseurs / mutateurs que quand cela est exigé par le travail d'implémentation
- Pensez vos classes comme des fournisseurs de services et pas comme des structures de données
- Objectif : réduire le couplage (*tight coupling*)

Règle 2. Encapsuler ce qui varie dans une hiérarchie spécifique

Règle 3. Programmer pour une interface et non pour une implémentation

Règle 4. Privilégier la composition à l'héritage

Contre-exemple : le patron Messager

- Un objet qui agit comme une simple structure de données
 - Ses données sont **publiques**
- Quand ?
 - Les données et leur représentation sont la raison d'être de l'objet
- Exemple 1 :
 - Un **Point 2D** est intrinsèquement caractérisé par ses deux attributs x et y. Donc, les attributs font la classe et il n'y a pas de raison de les encapsuler

```
public class Point2D {  
    public int x;  
    public int y;  
}
```
- Exemple 2 :
 - Transformer une liste d'arguments d'une méthode par un seul objet.
 - ▶ $f(a,b,c) \rightarrow f(o)$ avec o.a, o.b, o.c
 - ▶ Permet d'augmenter le nombre de paramètres sans modifier la signature de la méthode

Règles de conception

Règle 1. Réduire l'accessibilité des membres de classe

Règle 2. Encapsuler ce qui varie dans une hiérarchie spécifique

Règle 3. Programmer pour une interface et non pour une implémentation

Règle 4. Privilégier la composition à l'héritage

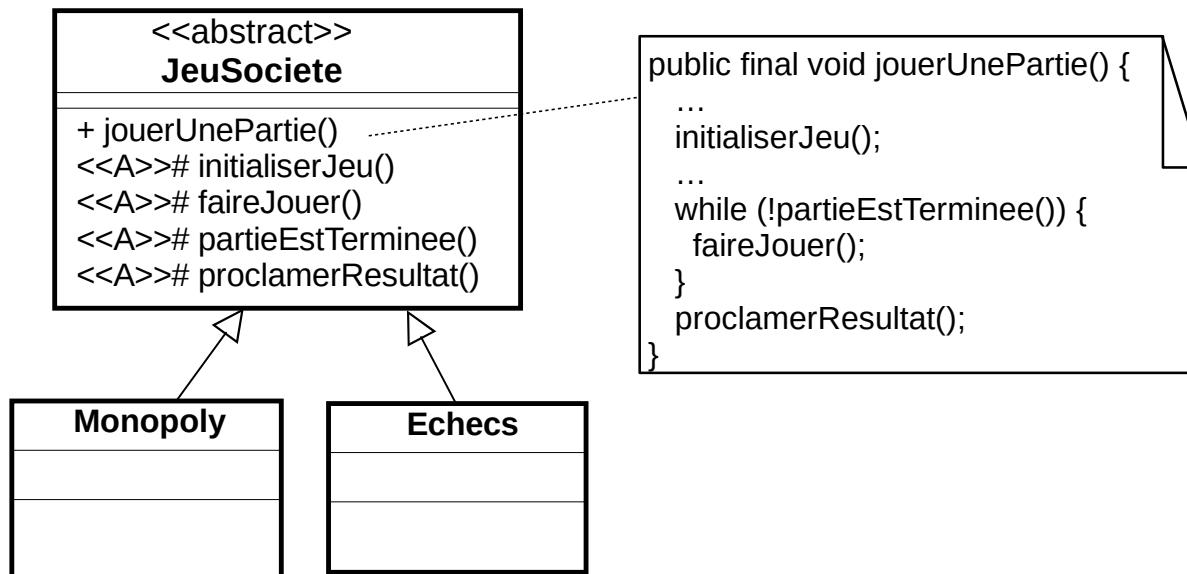
Quiz : patron

- Quel patron de conception permet d'encapsuler les variations suivantes :
 - variation du comportement **d'un service** d'un objet en fonction de ses **propriétés** ?
 - ▶ Décorateur
 - variation du comportement **des services** d'un objet en fonction de son état ?
 - ▶ État
 - variation de **parcours** d'un agrégat ?
 - ▶ Itérateur
 - variation de la liste des **objets dépendant** d'un objet de référence ?
 - ▶ Observateur
 - variation de la **liste des méthodes** d'une classe ?
 - ▶ Visiteur
 - variation de **l'algorithme** d'un service ?
 - ▶ Stratégie
 - variation **d'une partie de l'algorithme** d'un service ?
 - ▶ Patron de méthode

Patron de méthode (Template method)

14

- **Quand ?**
 - On souhaite une méthode avec des parties dépendantes d'un type d'objet
- **Ce qui varie**
 - Les parties d'un algorithme
- **Solution**



Quiz : patron

- Quel patron de conception permet de capturer les variations suivantes :
 - variation de comportement **d'un service** d'un objet en fonction de ses **propriétés** ?
 - ▶ Décorateur
 - variation du comportement **des services** d'un objet en fonction de son état ?
 - ▶ État
 - variation de **parcours** d'un agrégat ?
 - ▶ Itérateur
 - variation de la liste des **objets dépendant** d'un objet de référence ?
 - ▶ Observateur
 - variation de **l'algorithme** d'un service ?
 - ▶ Stratégie
 - variation de la **liste des méthodes** d'une classe ?
 - ▶ Visiteur
 - variation **d'une partie de l'algorithme** d'un service ?
 - ▶ Patron de méthode
 - variation sur l'objet d'une chaîne capable de répondre à une requête
 - ▶ Chaîne de responsabilité

Le patron Chaîne de responsabilité

- **Quand ?**

- Déléguer la réalisation d'une fonctionnalité à une chaîne de délégation inconnue (cf. le tondeur de pelouse).

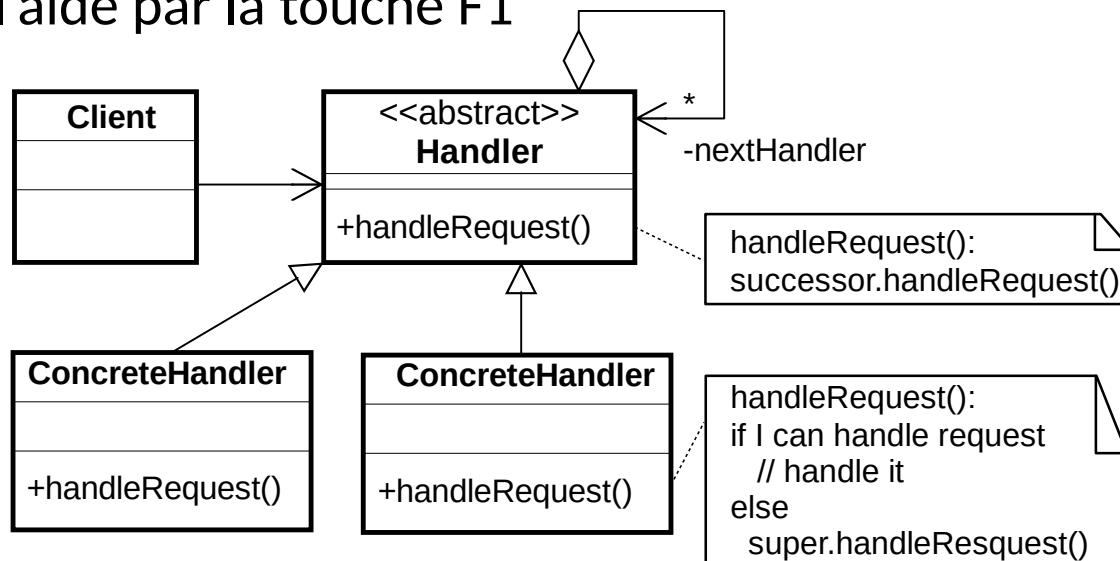
- **Ce qui varie**

- L'objet qui peut répondre à un service

- **Solution**

Créer une chaîne d'objet susceptibles de répondre à la requête.

- **Exemple : accès à l'aide par la touche F1**



Règles de conception

Règle 1. Réduire l'accessibilité des membres de classe

Règle 2. Encapsuler ce qui varie dans une hiérarchie spécifique

Règle 3. Programmer pour une interface et non pour une implémentation

- On remarque ainsi qu'il y a toujours une classe abstraite ou une interface à la racine du graphe d'héritage des patrons.
- Le code doit être le plus générique possible en référençant les objets avec la classe la plus haute possible dans la hiérarchie :
 - ▶ p.ex. JeuSociete jeu = new Monopoly();
jeu.jouerUnePartie();
- Motivation : être moins dépendant des évolutions qui interviennent le plus souvent sur les implémentations.

Règle 4. Privilégier la composition à l'héritage

Règles de conception

Règle 1. Réduire l'accessibilité des membres de classe

Règle 2. Encapsuler ce qui varie dans une hiérarchie spécifique

Règle 3. Programmer pour une interface et non pour une implémentation

Règle 4. Privilégier la composition à l'héritage

- L'héritage est très souvent une erreur de conception.
- Il faut transformer la création statique d'objet composite (qui utilise l'héritage) par la construction dynamique (qui utilise la composition).
 - ▶ `ie new Classe1Classe2() → new Classe1(new Classe2());`
- Ne jamais dériver de classes concrètes : risque de violer le principe de substitution de Liskow

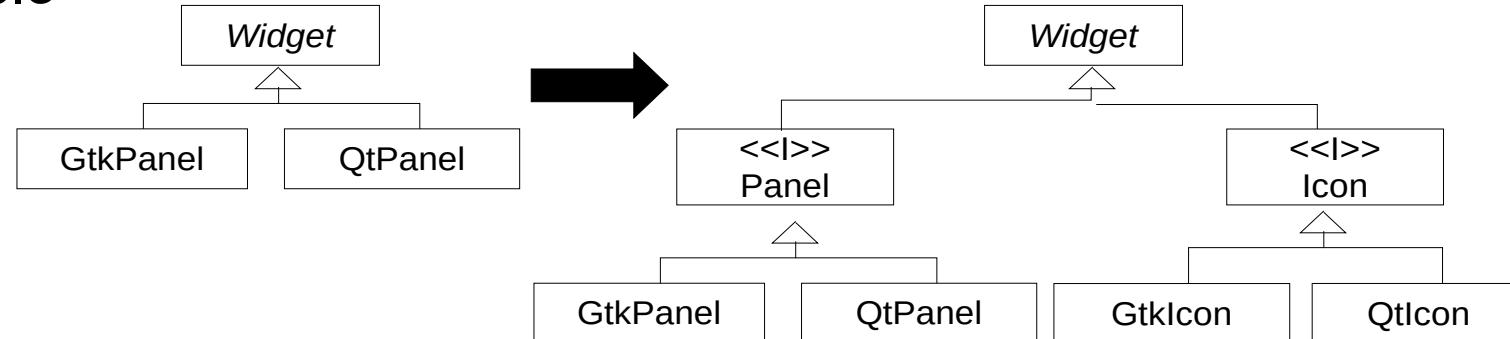
Quiz : patron

- Quel patron permet d'éviter de créer autant de classes que de combinaison de propriétés d'un objet pour réaliser un service ?
 - Décorateur
 - ▶ `objet = new Decorateur1(new Decorateur2(new ClasseBase()));`
- Quel patron permet d'éviter de créer autant de classes que la combinaison des abstractions par les implémentations ?
 - Pont

Patron de conception Pont

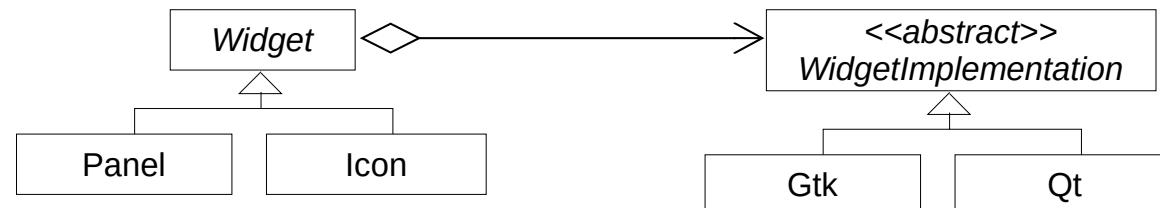
20

- Exemple



- Construction d'une instance :
 - ▶ `Widget panel = new GtkPanel();`
 - ▶ Problème explosion combinatoire du nombre de classes

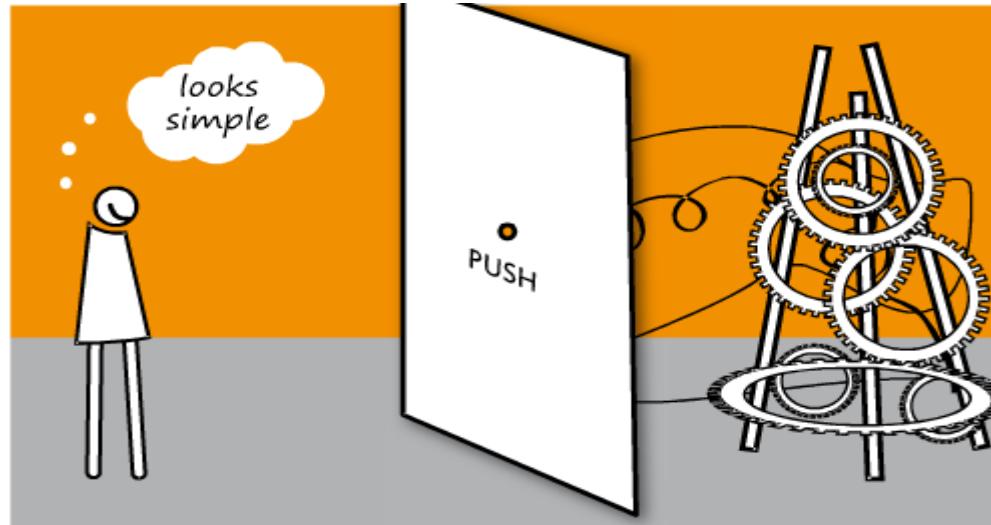
- **Solution** : Découpler les variations en abstraction et en implémentation pour éviter de créer des classes par leur combinaison



- Construction d'une instance :
 - ▶ `Widget panel = new Panel(new Gtk());`

Dernier patron : Le patron Façade

- Fournir une interface qui rend un sous-système plus facile à utiliser.



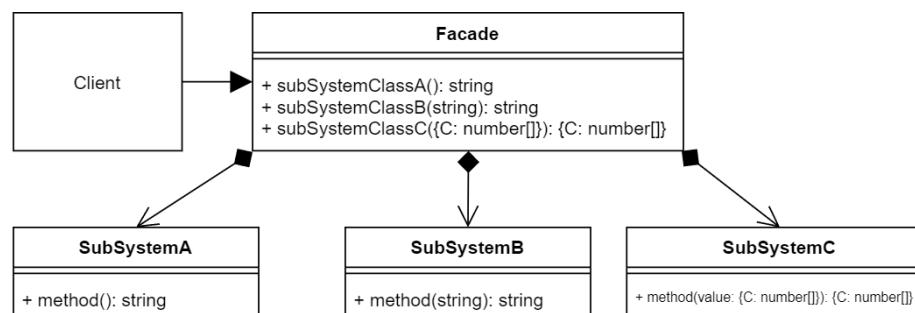
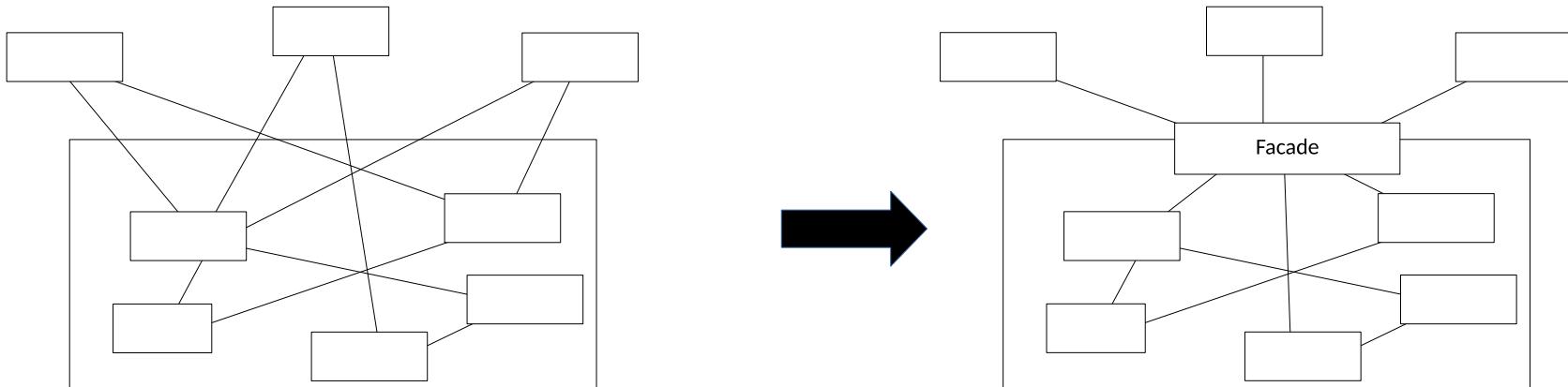
Le patron Façade

- **Quand ?**

- On souhaite simplifier l'interface d'accès à un sous-système

- **Solution**

- Une classe abstraite qui regroupe les éléments de l'interface
 - Cf. Principe de ségrégation des interfaces



Anti-patrons

- Les anti-patrons sont des erreurs courantes dans la conception des logiciels.
- Les anti-patrons se caractérisent souvent par une lenteur excessive du logiciel, des coûts de réalisation et de maintenance élevés, des comportements anormaux et l'introduction de bugs.

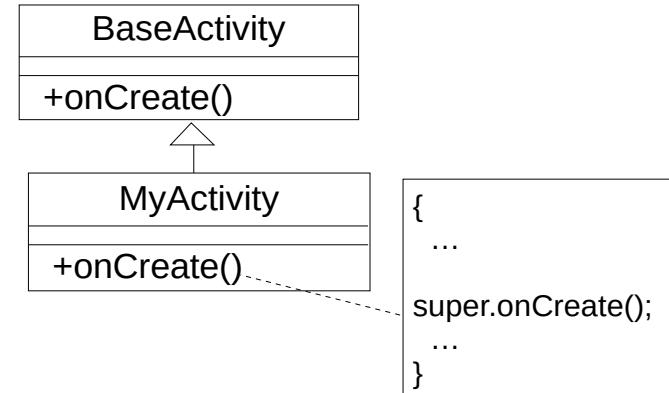
Anti-pattern

24

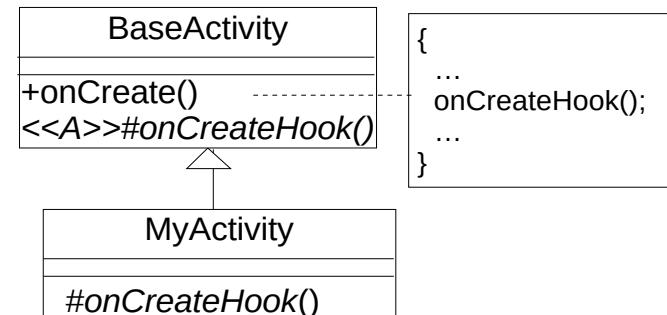
- Call super

- Mettre une méthode de base obligatoire dans la classe abstraite que les sous-classes peuvent compléter mais sans les obliger à appeler la méthode de base.

Exemple : Activity du kit de développement Android



- Solution : patron de conception Patron de méthode (*Template Method*)



- Remarque : pour résoudre ce problème, Android utilise l'annotation `@CallSuper` sur les méthodes (eg, `onCreate()`, `onResume()`, `onDestroy()`) → Voir cours de développement mobile)

- Patrons de conception
 - Les patrons ont quelques fois des coûts supplémentaires.
 - Par contre, d'autres sont des solutions sine qua non (adaptateur, commande, décorateur, visiteur...).
- Attention : pour chaque règle, il existe des cas où son application serait une pure folie.
 - e.g raison de sécurité, de performances, etc.
- Contrairement à une première impression, les patrons de conception se marient très bien avec les approches Agiles.
 - Le principe YAGNI suggère de commencer par la solution la plus simple n'intégrant pas forcément de patrons de conception.
 - Ensuite, le développement itératif incite à se poser la question d'une refonte de la conception reposant sur les patrons pour prendre en compte de nouvelles fonctionnalités réutilisant des parties du code existant.
 - ▶ Ne pas accumuler de dettes techniques.
 - Recherche de l'excellence technique.