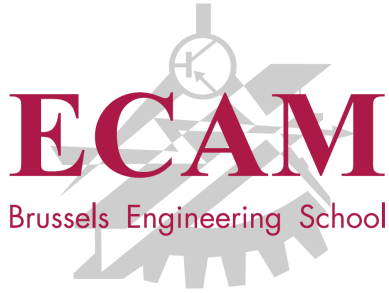


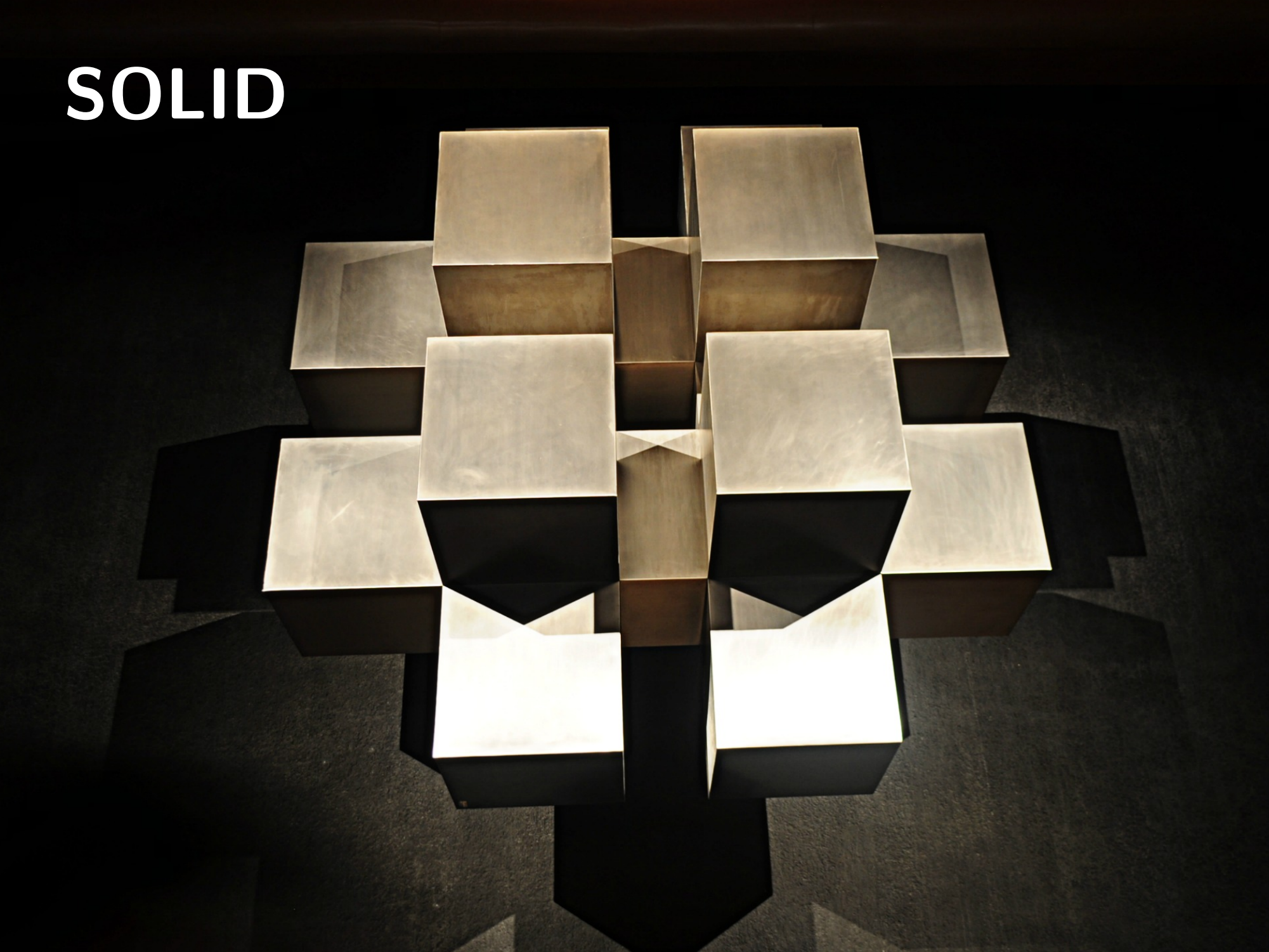
E303B Programmation orientée objet

Séance 7

Principes de l'orienté objet

Sébastien Combéfis, Quentin Lurkin

SOLID



SOLID

- Cinq principes en design d'applications avec l'orienté objet

  Par Michael Feathers pour les principes de Robert C. Martin

- Permet une application plus facilement maintenable

  Guidelines qui permettent de nettoyer le mauvais code

- Les cinq principes à retenir avec l'acronyme SOLID

  S ingle Responsibility Principle (SRP)

  O pen/closed principle (OCP)

  L iskov Substitution Principle (LSP)

  I nterface Segregation Principle (ISP)

  D ependency Inversion Principle (DIP)

Single responsibility principle

A class should have one and only one reason to change, meaning that a class should only have one job.

Principe de responsabilité unique

- Une classe/un module doit avoir une responsabilité unique

  Une partie de la fonctionnalité globale de l'application

- La responsabilité doit être encapsulée dans la classe/module

  Tous les services de la classe sont liés à sa responsabilité

- Facilite l'évolution et le changement d'un code

  Les endroits où changer sont facilement localisables

« *A class should have only one reason to change.* » — Robert C. Martin

Violation de SRP (1)

- Application qui permet de générer une facture

  Récupération de données depuis une BDD et mise en page

- Deux fonctionnalités à séparer
  - `Bill` représente la facture et son contenu
  - `BillFormatter` permet de créer un rendu de la facture

- On peut même prévoir plusieurs mises en page

  Par exemple `PDFBillFormatter` qui étend `BillFormatter`

Violation de SRP (2)

```
1  public class Circle : Shape
2  {
3      public double radius;
4  }
5
6  public class Square : Shape
7  {
8      public double length;
9  }
10
11 public class AreaCalculator
12 {
13     private readonly IEnumerable<Shape> shapes;
14
15     public double Sum()
16     {
17         // [...]
18     }
19
20     public void OutputResult()
21     {
22         // [...]
23     }
24 }
```

→ Créer un `SumCalculatorOutputter` avec paramètre `AreaCalculator`

Open-closed Principle

Objects or entities should be open for extension,

but closed for modification.

Principe ouvert/fermé

- Entités ouvertes à l'extension et fermée à la modification

  Classe, module, méthode...

- Autoriser la modification sans avoir accès au code

  Préserve l'encapsulation tout en restant ouvert

- Utiliser l'héritage et les interfaces/classes abstraites

Violation de OCP (1)

- Classe permettant de dessiner des formes

  Plusieurs formes sont supportées

- Pas facile d'ajouter du support pour de nouvelles formes

  Il faut créer une nouvelle classe et modifier `GraphicsEditor`

```
1  public class GraphicsEditor
2  {
3      public void drawShape (Shape s) { /* [...] */ }
4
5      private void drawCircle (Circle c) { /* [...] */ }
6
7      private void drawRectangle (Rectangle r) { /* [...] */ }
8  }
```

Violation de OCP (2)

- Déplacement de `draw` dans `Shape` pour ouvrir

  Et maintenir `GraphicsEditor` fermé

- Ajout du support pour une nouvelle forme très facile

  Il faut créer une nouvelle classe et y implémenter `draw`

```
1  public class GraphicsEditor
2  {
3      public void drawShape (Shape s)
4      {
5          s.draw();
6      }
7  }
8
9  public interface Shape
10 {
11     void draw();
12 }
```

Violation de OCP (3)

```
1  public double Sum()
2  {
3      double sum = 0;
4
5      foreach (Shape shape in shapes)
6      {
7          if (shape.GetType() == typeof (Square))
8          {
9              sum += Math.pow (shape.length, 2);
10         }
11         else if (shape.GetType() == typeof (Circle))
12         {
13             sum += Math.PI * Math.pow (shape.radius, 2);
14         }
15     }
16
17     return sum;
18 }
```

→ Ajouter une méthode `Area` dans `Shape` en bougeant responsabilité de calcul

Liskov substitution principle

Let q(x) be a property provable about objects of x of type T.

Then q(y) should be provable for objects y of type S

where S is a subtype of T.

Principe de Substitution de Liskov (1)

- Principe de Substitution de Liskov (LSP)

  Définit ce qu'est un bon sous-type, permettant l'héritage

- Soit S un sous-type de T
  - Tout objet de type T peut être remplacé par un de type S
  - Pas d'altération des propriétés désirables du programme

- Cas particulier de la relation de sous-typage

  Relation entre un sous-type et un super type par substituabilité

Behavioural subtyping

- Relation sémantique plutôt que seulement syntaxique

  Garantie d'interopérabilité sémantique dans une hiérarchie

- Définition formelle succincte du principe

  Extension de la logique de Hoare

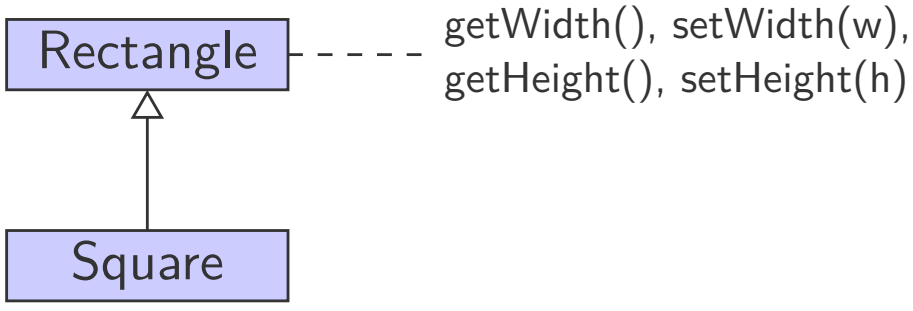
Liskov Substitution Principle (LSP)

Si $q(x)$ est une propriété démontrable pour tout objet x de type T,
Alors $q(y)$ doit est vraie pour tout objet y de type S
où S est un sous-type de T.

Violation de LSP

@post width et height
librement modifiables

@post width et height
doivent être égaux



- `Square` ne peut pas être utilisé partout à la place de `Rectangle`

  Car `setWidth` ou `setHeight` pourraient être appelées

- Redéfinition des mutateurs dans `Square` avec vérification

  Violation de la postcondition des `Rectangle`

Interface segregation principle

A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.

Principe de la ségrégation d'interface

- Pas forcer un client à dépendre de code pas utilisé

  Découpe d'une grosse interface en plus petites

- Le système reste le plus découplé possible

  Facilité la maintenance et l'évolution d'un programme

- Utilisation d'interfaces et classes abstraites

Violation de ISP (1)

- Logiciel pour une imprimante multi-fonctions Xerox

  Notamment impression de documents et agrafage

- Une seule classe `Job` pour gérer tous les types de job

  Méthodes de la classe appelées par toutes les fonctions

- Découpage des fonctions en interfaces `PrintJob`, `StapleJob`...

  Toutes implémentées par `Job`

Violation de ISP (2)

```
1  public interface Shape
2  {
3      double Area();
4
5      double Volume();
6  }
```

→ Découper en deux interfaces `Shape` et `SolidShape`

Dependency Inversion principle

Entities must depend on abstractions not on concretions.

It states that the high level module

must not depend on the low level module,

but they should depend on abstractions.

Principe d'inversion des dépendances

- Forme de découplage de modules dans un logiciel

  Module haut-niveau indépendant de l'implémentation bas-niveau

- Deux règles
  - Module haut-niveau dépend d'abstraction module bas-niveau
  - Abstraction dépend pas détails, détails dépendent abstractions

- Utilisation d'interfaces et classes abstraites

  Ajout d'un niveau abstrait entre les classes précédemment liées

Violation de DIP (1)

- Une classe `Manager` qui gère des travailleurs

  Au départ, un seul type de travailler `Worker`

- Dépendance directe entre les deux classes

  Module de haut-niveau dépendant de l'implémentation bas-niveau

```
1  public class Manager
2  {
3      private Worker worker;
4
5      public void setWorker (Worker w)
6      {
7          this.worker = worker;
8      }
9
10     public void manage()
11     {
12         worker.work();
13     }
14 }
```

Violation de DIP (2)

- Ajout d'une interface représentant les modules bas-niveau

  Module de haut-niveau dépend de l'interface

- Changement du type de worker facile, sans changer le `Manager`

```
1  public interface IWorker
2  {
3      void work();
4  }
5
6  public class Manager
7  {
8      private IWorker worker;
9
10     public void setWorker (IWorker w)
11     {
12         this.worker = worker;
13     }
14
15     // [...]
16 }
```

Violation de DIP (3)

```
1  public class PasswordReminder
2  {
3      private $dbConnection;
4
5      public function __construct (MySQLConnection $dbConnection)
6      {
7          $this->dbConnection = $dbConnection;
8      }
9  }
```

→ Ajout d'une interface `DBConnectionInterface` pour la connexion


STUPID
FinePix

STUPID

- Don't be STUPID but be SOLID

  Qui n'a jamais écrit du code stupide ? Mais il faut arrêter !

- Les six principes à retenir avec l'acronyme STUPID

  S ingleton

  T ight Coupling

  U ntestability

  P remature Optimization

  I ndescriptive Naming

  D uplication

Singleton

- Singleton parfois considéré comme un anti-pattern

  Pas un problème en soi, mais symptôme d'éventuels problèmes

- Deux majeures difficultés avec le singleton
  - Difficulté à tester un programme avec un état global
  - Dépendance à état global cache leurs dépendances

- Éviter, lorsque c'est possible, les membres de classe

Tight Coupling

- Couplage fort généralise le problème du singleton

  Mesure du degré de dépendance envers les autres modules

- Minimisation du couplage entre modules

  Changer un module ne doit pas nécessiter d'en changer d'autres

- Très difficile à réutiliser, et aussi à tester

Untestability

- Tester un programme ne doit pas être difficile

  Pas de tests unitaires car on ne prend pas le temps

- Code fortement couplé très difficile à tester

  Utilisation d'interfaces, d'abstraction...

Premature Optimization

- Optimiser à l'avance n'aura que des couts, aucun bénéfice

  Systèmes très complexes, pleins d'erreurs potentielles

- Deux règles de l'optimisation d'un programme

  1. Don't do it ;
  2. (for experts only !) Don't do it yet.

« *Premature optimization is the root of all evil.* » — Sir Tony Hoare*

* Popularisé par Donald E. Knuth

Indescriptive Naming

- Noms appropriés pour classes, méthodes, attributs, variables...

  Et ne pas faire d'abréviations !

- Les langages de programmation sont pour les humains

  On ne code pas pour l'ordinateur, il ne comprend que les 0 et 1...

Duplication

- Be lazy the right way - write code only once !
  - Don't Repeat Yourself (DRY)
  - Keep It Simple, Stupid (KISS)
- Attention à tous les types de duplications

  De code, de données, de logique, de design...