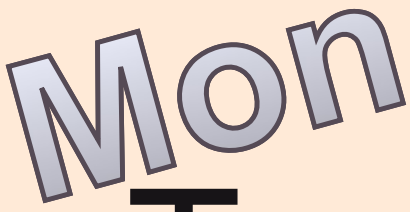


Top 5 des meilleures façons d'améliorer votre code

Eric De Carufel

Architecte Principal – Backend

GSoft → workleap

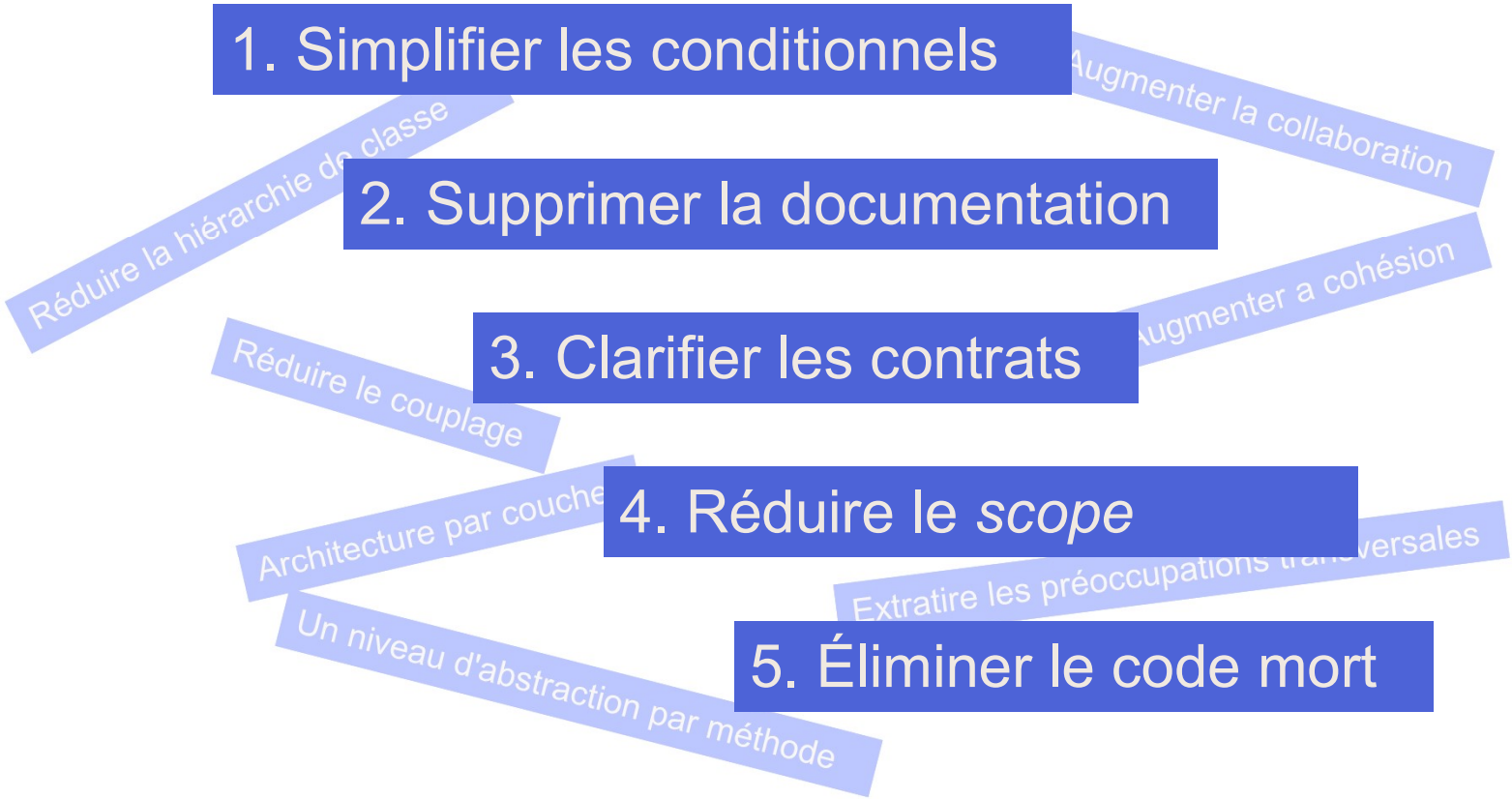
« Rendre les bonnes pratiques plus faciles à appliquer que les mauvaises. »

Introduction

→ « Legacy code is code without tests »
  → Michael Feather dans Working effectively with legacy code

→ Sans une maintenance constante, le code se dégrade rapidement

→ Nous devons détecter et éliminer les "code smells"


1. Simplifier les conditionnels
Augmenter la collaboration
Réduire la hiérarchie de classe
2. Supprimer la documentation
Augmenter a cohésion
3. Clarifier les contrats
Réduire le couplage
Architecture par couche
4. Réduire le *scope*
Extratire les préoccupations transversales
Un niveau d'abstraction par méthode
5. Éliminer le code mort

Simplifier les conditionnels – Pourquoi?

→Réduit la complexité

→Améliore la lisibilité

→Améliore la maintenabilité

→Améliore la réutilisabilité

1
2
3
4
5

Simplifier les conditionnels– Quand?

→il y a plus d'une condition (and / or);

→il y a trop de code dans le body;

→la condition est basée sur un type;

→il y a des *if* imbriqués

→il y a plusieurs décisions basées sur la même information (if / else if / switch case)

Simplifier les conditionnels – Comment?

- → Refactor conditional statement
  - → Decompose conditional
  - → Consolidate conditional expression
  - → Consolidate duplicate conditional fragments
  - → Introduce null object
  - → Flatten nested if
  - → Don't use negative
  - → Keep conditional statement lean
- → Avoid conditional statement
  - → Replace conditional with polymorphism
  - → Replace conditional logic with strategy
  - → Replace conditional dispatcher with command

1
2
3
4
5

SIMPLIFIER LES CONDITIONNELS

1
2
3
4
5

Decompose conditional

```
double CalcCharge(DateTime date, double quantity)
{
  double charge;

  if (date.DayOfYear < SummerStart || date.DayOfYear > SummerEnd)
    charge = quantity * this._winterRate + this._winterServiceCharge;
  else
    charge = quantity * this._summerRate;

  return charge;
}
```

```
double CalcCharge(DateTime date, double quantity)
{
  double charge;

  if (IsNotSummer(date))
    charge = quantity * this._winterRate + this._winterServiceCharge;
  else
    charge = quantity * this._summerRate;

  return charge;
}

static bool IsNotSummer(DateTime date)
{
  return date.DayOfYear < SummerStart || date.DayOfYear > SummerEnd;
}
```

SIMPLIFIER LES CONDITIONNELS

1
2
3
4
5

Consolidate conditional expression

```
double DisabilityAmount()
{
  if (this._seniority < 2) return 0;
  if (this._monthDisabled > 12) return 0;
  if (this._isPartTime) return 0;

  return this._amount * this._disabilityRate;
}
```

```
double DisabilityAmount()
{
  if (IsNotEligibleForDisability()) return 0;

  return this._amount * this._disabilityRate;
}

bool IsNotEligibleForDisability()
{
  return this._seniority < 2 ||
         this._monthDisabled > 12 ||
         this._isPartTime;
}
```

SIMPLIFIER LES CONDITIONNELS

1
2
3
4
5

Consolidate duplicate conditional fragments

```
void UpdateTotalPrice()
{
  if (IsSpecialDeal())
  {
    this._totalPrice = this._price * GetDealRate();
    Save();
  }
  else
  {
    this._totalPrice = this._price * GetRegularRate();
    Save();
  }
}
```

```
void UpdateTotalPrice()
{
  if (IsSpecialDeal())
  {
    this._totalPrice = this._price * GetDealRate();
  }
  else
  {
    this._totalPrice = this._price * GetRegularRate();
  }

  Save();
}
```

Introduce null object

```
void Display(Customer? customer)
{
    if (customer == null)
    {
        Console.WriteLine("no name");
    }
    else
    {
        Console.WriteLine(customer.Name);
    }
```

```
public class Customer
{
    public Customer(string name)
    {
        this.Name = name;
    }

    public string Name { get; }
}
```

```
public class NullCustomer : Customer
{
    public NullCustomer() : base("no name")
    {
    }
}
```

```
void Display(Customer customer)
{
    Console.WriteLine(customer.Name);
}
```

Flatten nested if

```
double GetPayAmount()
{
  double result;
  if (this._isDead) result = DeadAmount();
  else
  {
    if (this._isSeparated) result = SeparatedAmount();
    else
    {
      if (this._isRetired) result = RetiredAmount();
      else result = NormalPayAmount();
    }
  }

  return result;
}
```

```
double GetPayAmount()
{
  if (this._isDead) return DeadAmount();
  if (this._isSeparated) return SeparatedAmount();
  if (this._isRetired) return RetiredAmount();
  return NormalPayAmount();
}
```

SIMPLIFIER LES CONDITIONNELS

1
2
3
4
5

Don't use negative

```
void Process(string? state)
{
    if (!string.IsNullOrEmpty(state) && state ≠ "RUNNING")
    {
        // ...
    }
}
```

```
void Process(string? state)
{
    if (CanProcess(state))
    {
        // ...
    }
}

static bool CanProcess(string? state)
{
    return IsAvailable(state) && IsReady(state);
}

static bool IsAvailable(string? state)
{
    return !string.IsNullOrEmpty(state);
}

static bool IsReady(string? state)
{
    return state ≠ "RUNNING";
}
```

Keep conditional statement lean

- → Autant que possible, avoir une seule condition
  - → Utilisez une méthode pour combiner plusieurs conditions

- → Inverser le if si la plupart (ou tout) le code se trouve dans la branche true
  - → Attention à la double négation

SIMPLIFIER LES CONDITIONNELS

1
2
3
4
5

Replace conditional with polymorphism

```
double GetSpeed(VehiculeType vehiculeType)
{
  switch (vehiculeType)
  {
    case VehiculeType.Car:
      return GetBaseSpeed();
    case VehiculeType.Truck:
      return GetBaseSpeed()
    case VehiculeType.Plane:
      return GetBaseSpeed()
    default:
      throw new ArgumentOutO
        nameof(vehiculeType)
  }
}
```

```
abstract class Vehicule
{
  protected abstract double GetSpeed();
}

class Car : Vehicule
{
  protected override double GetSpeed() => GetBaseSpeed();
}

class Truck : Vehicule
{
  protected override double GetSpeed() => GetBaseSpeed() * LoadFactor();

  private double LoadFactor()
  {
    throw new NotImplementedException();
  }
}

class Plane : Vehicule
{
  protected override double GetSpeed() => GetBaseSpeed() * WindDragFactor();

  private double WindDragFactor()
  {
    throw new NotImplementedException();
  }
}
```

SIMPLIFIER LES CONDITIONNELS

1
2
3
4
5

Replace conditional logic with strategy

```
class Loan
{
  public double Capital()
  {
    var result = 0.0;
    // ...
    // many different implementations
    // ...
    return result;
  }
}
```

```
class CapitalStrategy1 : CapitalStrategy
{
  publ
  {
    va
    //
    re
  }
}
```

```
class CapitalStrategy2 : CapitalStrategy
{
  public override double Capital(Loan loan)
  {
    var
    // I
    retu
  }
}
```

```
class CapitalStrategy3 : CapitalStrategy
{
  public override double Capital(Loan loan)
  {
    var result = 0.0;
    // Implementation of strategy 3
    return result;
  }
}
```

```
class Loan
{
  private readonly CapitalStrategy _strategy;

  Loan(CapitalStrategy strategy)
  {
    this._strategy = strategy;
  }

  double Capital()
  {
    return this._strategy.Capital(this);
  }
}

abstract class CapitalStrategy
{
  public abstract double Capital(Loan loan);
}
```

SIMPLIFIER LES CONDITIONNELS

1
2
3
4
5

Replace conditional dispatcher with command

```
void ExecuteAction(string actionName)
{
  if (actionName == "Save")
  {
    // lots of code
  }
  else if (actionName == "Delete")
  {
    // lots of code
  }
  // Other if else statements
}
```

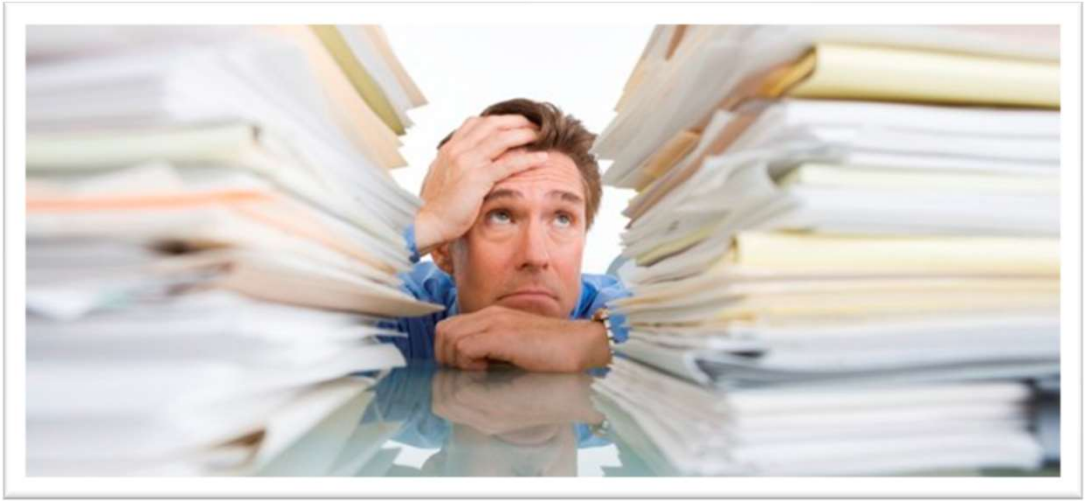
```
readonly Dictionary<string, ICommand> _commands = new();

void BuildCommands()
{
  this._commands.Add("Save", new SaveCommand());
  this._commands.Add("Delete", new DeleteCommand());
}

ICommand GetCommand(string actionName)
{
  return this._commands[actionName];
}

void ExecuteAction(string actionName)
{
  var command = GetCommand(actionName);
  command.Execute();
}
```

Supprimer la documentation – Pourquoi?

→Améliore la lisibilité

→Améliore la maintenabilité

→Évite les commentaires désuets



Supprimer la documentation – Quand?

- → Chaque fois qu'un commentaire est autre chose que de l'information (utile), une **intention**, une **clarification**, un **avertissement**, un **TODO** ou une **amplification**.
- → Le commentaire décrit, ligne par ligne, le code
  - → Exemple:
    - → // Getting connection string from configuration
    - → // Opening connection
    - → // Retrieving data
    - → // Closing connection

Supprimer la documentation – Comment?

- → Remplacez les commentaires par une bonne nomenclature
  - → Extract method
  - → Utilisez des noms significatifs
- → Écrivez des commentaires utiles
- → Respectez la nomenclature (MSDN: Guidelines for names)
  - → Properties
  - → Enums
  - → Events
  - → Methods

SUPPRIMER LA DOCUMENTATION

1
2
3
4
5

Extract method

```
void RenderMain(int width, int height)
{
  // Render Header
  Console.WriteLine("Random Table");

  Console.Write("    ");

  for (int col = 1; col ≤ width; col++)
    Console.Write($"{col,4}");

  Console.WriteLine();

  // Redner Rows
  for (int row = 1; row ≤ height; row++)
  {
    Console.Write($"{row,4}");

    // Rendoer Columns
    var random = new Random();
    for (int col = 1; col ≤ width; col++)
      Console.Write($"{random.Next(100),4}");
    Console.WriteLine();
  }

  Console.WriteLine();
}
```

```
void RenderMain(int width, int height)
{
  RenderHeader(width);
  RenderRows(width, height);
}
private static void RenderHeader(int width)
{
  Console.WriteLine("Random Table");

  Console.Write("    ");

  for (int col = 1; col ≤ width; col++)
    Console.Write($"{col,4}");

  Console.WriteLine();
}
private static void RenderRows(int width, int height)
{
  for (int row = 1; row ≤ height; row++)
  {
    Console.Write($"{row,4}");

    RenderColumns(width);
  }
}
private static void RenderColumns(int width)
{
  var random = new Random();
  for (int col = 1; col ≤ width; col++)
    Console.Write($"{random.Next(100),4}");
  Console.WriteLine();
}
```

SUPPRIMER LA DOCUMENTATION

1
2
3
4
5

Utilisez des noms significatifs

- → Nommez selon l'intention / évitez le mapping
  - → d → elapsedDays
- → Évitez la désinformation
  - → AccountList → Accounts
- → Assurez-vous d'avoir une distinction significative
  - → amount → invoiceTotal, invoiceSubTotal
- → Utilisez des noms prononçables
  - → genymdhms → generationTimestamp
- → Utilisez un nom pour nommer une classe
- → Utilisez un verbe pour nommer une méthode
- → Ne soyez pas créatif, utilisez les noms standards
  - → Destroy, Kill, Obliterate → Delete
- → Domaine de solution ↔ domaine du problème

SUPPRIMER LA DOCUMENTATION

1
2
3
4
5

Properties

→ Utilisez le PascalCase naming

→ Nommez les propriétés avec un nom ou un adjectif

→ N'utilisez pas de noms qui pourraient être confondu avec une méthode Get

→ Préfixez les booléen avec Can, Is ou Has

```
public string InvoiceNumber { get; set; } = "";
public bool IsPaid { get; }
```

SUPPRIMER LA DOCUMENTATION

1
2
3
4
5

Enums

→ Considérez le premier élément comme valeur par défaut

→ Utilisez le PascalCasing naming

→ Les Enum simples doivent utiliser le singulier

→ Les Enum de type Bit fields devraient utiliser le pluriel et avoir l'attribut Flags

→ Les valeurs des Enum de type bit fields doivent être cohérentes (Read & Write == ReadWrite)

```
public enum SystemState
{
    Idle,
    Working,
    Faulted
}
```

```
[Flags]
public enum FilePermissions
{
    None = 0b0000, // 0x00
    Read = 0b0001, // 0x01
    Write = 0b0010, // 0x02
    ReadWrite = 0b0011, // 0x03
    Delete = 0b0100, // 0x04
    Manage = 0b1000 // 0x08
}
```

SUPPRIMER LA DOCUMENTATION

1
2
3
4
5

Events

→Utilisez le *PascalCase naming*

→Nommez les événement avec un verbe au *present progressive* pour les pré-événements et au passé pour les post-événements

→Fournissez une version *virtual* de l'événement

→Fournissez une façon d'annuler un pré-événement

SUPPRIMER LA DOCUMENTATION

1
2
3
4
5

Events

```
public event EventHandler<SavingEventArgs>? Saving;

public event EventHandler<SavedEventArgs>? Saved;

public void Save()
{
  if (!OnSaving(new SavingEventArgs())) return;
  Save();
  OnSaved(new SavedEventArgs());
}

protected virtual bool OnSaving(SavingEventArgs savingEventArgs)
{
  Saving?.Invoke(this, savingEventArgs);
  return !savingEventArgs.Cancel;
}

protected virtual void OnSaved(SavedEventArgs savedEventArgs)
{
  Saved?.Invoke(this, savedEventArgs);
}
```

```
public class SavingEventArgs : CancelEventArgs
{
}

public class SavedEventArgs : EventArgs
{
}
```

SUPPRIMER LA DOCUMENTATION

1
2
3
4
5

Methods

- → Utilisez des verbes pour nommer les méthodes
  - → ProcessPayment
- → Exprimez clairement le retour attendu lors de l'appel de la méthode
  - → CreateCustomer
  - → GetInvoice
- → Utilisez une nomenclatue uniforme (Get, Fetch ou Retrieve mais pas tous dans le même contexte)

Clarifier les contrats – Pourquoi?

→ Améliore la performance

→ Améliore la lisibilité

→ Améliore la reutilisabilité

Clarifier les contrats– Quand?

→ Il y a trop de paramètres (combien est-ce?)

→ Une méthode fait plus d'une chose

→ Une méthode utilise des paramètres out

→ Vous avez besoin de valeur par défaut

Clarifier les contrats– Comment?

- → Réduisez le nombre de paramètres
  - → Introduce parameter object
  - → Create overload with less parameters
  - → Use default value
- → Les ouputs
  - → La valeur de retour
  - → Les paramètres out
- → Overload dans le bon ordre

Introduce parameter object

```
public IEnumerable<Transaction> GetTransactions(DateTime start, DateTime end)
{
  var result = from t in this._transactions
    where t.Timestamp ≥ start
          && t.Timestamp < end
    select t;
  return result;
}
```

```
public class Range<T> where T : IComparable<T>
{
  private readonly T _start;
  private readonly T _end;

  public Range(T start, T end)
  {
    _start = start;
    _end = end;
  }

  public bool Contains(T target)
    ⇒ target.CompareTo(this._start) ≥ 0
       && target.CompareTo(this._end) < 0;

  // ...
```

```
public IEnumerable<Transaction> GetTransactions(Range<DateTime> range)
{
  var result = from t in _transactions
    where range.Contains(t.Timestamp)
    select t;
  return result;
}
```

thods:
nside, Outside, ...

Create overload with fewer parameters

```
public void Write(byte[] buffer,
    int length,
    int offset,
    byte pad)
{
    // Implementation
}
```

```
public void Write(byte[] buffer,
    int length,
    int offset,
    byte pad)
{
    // Implementation
}

public void Write(byte[] buffer,
    int offset,
    int length)
    => Write(buffer, length, offset, PAD_BYTE);

public void Write(byte[] buffer,
    int offset)
    => Write(buffer, buffer.Length, offset, PAD_BYTE);

public void Write(byte[] buffer)
    => Write(buffer, buffer.Length, 0, PAD_BYTE);
```

Use default value

```
public void Write(byte[] buffer,
    int length,
    int offset = 0,
    byte pad = PAD_BYTE)
{
    // Implementation
}
```

```
public void Write(byte[] buffer)
    => Write(buffer, buffer.Length, 0, PAD_BYTE);
```

Réduire le Scope – Pourquoi?

→Évite les effets de bords

→Améliore la réutilisabilité

→Améliore la maintenabilité

Réduire le Scope – Quand?

→Un field est utilisé par trop peu de méthodes

→Les membres publics exposent le comportement interne

Réduire le Scope – Comment?

- → Visibilité
  - → protected
  - → private
  - → internal
- → Responsabilité
  - → Déplacer un *field* vers une méthode
  - → Scinder une classe
  - → Déplacer une variable près de son utilisation
- → Durée de vie
  - → Créer les instances au besoin
  - → Éviter de garder des références inutiles

Éliminer le code mort – Pourquoi?

→Parcequ'il le faut
→Améliore la maintenabilité
→Améliore la performance
→Améliore la lisibilité
→Augmenter la couverture de test

Éliminer le code mort – Quand?

- → Vous savez que le code est mort
  - → Exemples:
    - → Code en commentaire
    - → Code innaccessible
- → Vous pensez que le code est mort
  - → Exemple:
    - → Du code qui semble ne rien faire d'utile
    - → Du vieux code qui n'a jamais changé depuis …
- → Vous voulez que le code soit mort
  - → Exemple:
    - → Cette espèce de méga méthode que personne veut toucher

Éliminer le code mort – Comment?

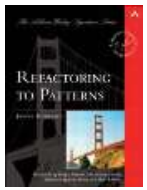
→ Identifiez et retirez le code mort
  → Effacez le code
  → Compilez
  → Roulez les tests

→ Qu'est-ce que du code mort?
  → Du code en commentaire
  → Toutes lignes de code non couverte par un test unitaire

→ Outils
  → Il existe des outils qui supprime automatiquement le code non couvert par au moins un test

References

→ Refactoring – Improving the design of existing code
- → Auteur: Martin Fowler
- → Edition: Addison Wesley
- → ISBN: 978-0-201-48567-7



→ Refactoring to patterns (Martin Fowler signature)
- → Auteur: Joshua Kerievsky
- → Edition: Adison Wesley
- → ISBN: 978-0-321-21335-1

→ Clean code – a handbook of agile software craftsmanship
- → Auteur: Robert C. Martin
- → Edition: Prentice Hall
- → ISBN: 978-0-132-35088-4

→ Working effectively with legacy code
- → Auteur: Michael C. Feather
- → Edition: Prentice Hall
- → ISBN: 978-0-13-117705-5

La fin

→ Rappelez-vous

Simplifier les conditionnels

Supprimer la documentation

Clarifier les contrats

Réduire le scope

Éliminer le code mort

→ Questions?