Complexité cyclomatique

Kerian Pelat, Thibault Walterspieler, Alex Mongeot, Damien Mathieu, Gregory Albouy et Thomas Moreira

Maintenabilité

Les enjeux de maintenabilité, de lisibilité, clarté et compréhensibilité grandissent au fil du temps.

Solutions en surcouche

Conventions de style et de nommage

Commentaires à valeur ajoutée

Documentation, READMEs, schemas et wikis

Tests et spécifications métier

Bien, mais insuffisant

```
func doSomething(i int) {
    if i < 1 {
        if i < 2 {
            if i < 3 {
                if i < 4 {
                    if i < 5 {
                        if i < 6 {
                        } else {
                        }
                    } else {
                    }
                } else {
                    if i < 5 {
                    } else {
                    }
                }
            } else {
                if i < 4 {
                    if i < 5 {
                    } else {
                    }
                } else {
                }
            }
        } else {
        }
    } else {
    }
}
```

Évaluer la complexité du code

Évaluer la complexité **cognitive** du code pour un **humain** de manière **objective**.

On utilise la complexité cylomatique.

Principe de calcul

Le nombre de **branches empruntables**.

La complexité devient plus importante pour chaque :

- structure de contôle (`if` , `else` , `for` , `while` , ...)
- condition booléenne (`!` , `&&` , `||`)

Scores (à titre d'exemple arbitraire)

Il faut maintenir la complexité le plus bas possible.

Complexité	Interprétation
1-10	structuré, compréhensible et testable trivialement
10-20	complexe mais acceptable selon les cas, tests non évidents
20-40	très complexe et trop difficilement testable, à refactorer
40+	intestable, inmaintenable et très certainement incompréhensible

Bad code et refactor

Exemples de code à haute complexité et propositions de refactoring possible.

Structures de contrôle nestées 💩

```
function sendEmail(to: string, message: string, attachement: File): void {
  if (isValidEmailAddress(to)) {
    if (isValidAttachement(attachement)) {
      if (message !== "") {
        Mailer.send(to, message, attachement)
      } else {
        throw new Error("Cannot send empty message")
      }
    } else {
      throw new Error("Attachement is not valid")
    }
  } else {
    throw new Error("Destination email is not valid")
  }
}
```

Trivial mais tout doit être lu pour être compris, gestion des exceptions douteuse.

Guards, early returns et happy path 👍

Pour répondre à ces limites, on peut refactorer de sorte à suivre un *happy path*, le comportement principal de la fonction reste au niveau d'indentation 1 :

```
function sendEmail(to: string, message: string, attachement: File): void {
  if (!isValidEmailAddress(to)) {
    throw new Error("Destination email is not valid")
  }
  if (!isValidAttachement(attachement)) {
    throw new Error("Attachement is not valid")
  }
  if (message === "") {
    throw new Error("Cannot send empty message")
  }

  Mailer.send(to, message, attachement)
}
```

Guards et early returns : gestion des cas d'exceptions lisible.

Happy path : niveau d'indentation à 1, valeur de retour attendu à la fin de la fonction.

Conditions complexes 💩

```
func isTestOk(hasPassed bool, shouldNotPass bool) bool {
  return (hasPassed && !shouldNotPass) || (!hasPassed && shouldNotPass)
}
```

Complexité = 3

```
A && !B // +1
!A && B // +1
X || Y // +1
```

Exclusion des conditions superflues 👍

```
func isTestOk(hasPassed bool, shouldNotPass bool) bool {
  return hasPassed != shouldNotPass
}
```

Complexité = 1

```
!= // +1
```

Routines complexes inline 💩

Extraction de fonctions 👍

Cascades de `if` `else if` 💩

```
function notifyBillingStatus(mailer: Mailer, user: User): void {
  let message = ""
  if (!user.active) { // +1
    message = "Your account is not active"
  } else if (user.paidAt && user.paidAt > 0) { // +2
    if (user.paidAt > months.ago(1)) {
      message = "You paid your bill this month"
    } else if (user.paidAt < months.ago(1) && user.paidAt > months.ago(2)) { // +2
      message = "You need to pay your bill this month"
    } else { // +1
      message = "You did not pay, your account is being deactivated"
    }
  } else { // +1
    message = "You need to pay your first bill to activate your account"
  }
  mailer.send(message, user)
}
```

`user.paidAt` est un timestamp Unix.

Complexité = 8

`switch` statements 👍

```
function notifyBillingStatus(mailer: Mailer, user: User): void {
  if (!user.active) { // +1
    mailer.send("Your account is not active", user)
    return
  }
  if (!user.paidAt || user.paidAt <= 0) { // +1
    mailer.send("You need to pay your first bill to activate your account", user)
    return
  }
  let message = ""
  switch (months.diff(user.paidAt, months.ago(1))) { // +1
    case 0:
      message = "You paid your bill this month"
      break
    case 1:
      message = "You need to pay your bill this month"
      break
    default:
      message = "You did not pay, your account is being deactivated"
      break
  }
  mailer.send(message, user)
}
```

Complexité = 4

Structures de données à la place de structure de contrôle

> structure de données + algorithme = programme

Remplacer du code **procédural** par du code **impératif**.

Strategy pattern : via map/dictionnaire 👍

```
const billingStatusBehaviors: { [k: BillingStatus]: NotifyFunction } = {
  paid: (mailer, user) => mailer.send("You paid your bill this month", user),
  late: (mailer, user) => mailer.send("You did not pay, ...", user),
  // ...
}

function getNotifyFunction(status: BillingStatus): NotifyFunction {
  return billingStatusBehaviors[status]
}

function notifyBillingStatus(status: BillingSatus, mailer: Mailer, user: User): void {
  const notify = getNotifyFunction(status)
  notify(mailer, user)
}
```

Centralise les définitions et les déplace hors du scope du code appelant.

Choix du comportement à utiliser encapsulé.

Strategy pattern : via polymorphisme (OOP) 👍

```
interface UserNotifier {
  notifyBillingStatus(): void
}

class UserPaidNotifier implements UserNotifier {
  notifyBillingStatus(): void {
    this.mailer.send("You paid your bill this month", this.user)
  }
}

class UserLateNotifier implements UserNotifier {
  notifyBillingStatus(): void {
    this.mailer.send("You did not pay, your account is being deactivated", this.user)
  }
}
// ...

function notifyBillingStatus(userNotifier: UserNotifier): void {
  userNotifier.notifyBillingStatus()
}
```

Associe les comportements à une structure de données spécialisée.

Abstrait l'implémentation hors du code appelant.

Sources regulières de complexité

Prompt à une forte complexité et inmaintenabilité. Attention particulièrement à :

- fonctions *top level* (`main()`, `run()` `exec()` et similaires)
- fonctions de configuration (lecture/écriture de variables globales ou hors scope)
- fonctions de validation ou d'assertion de valeur ou comportement métier (nœud à branches et conditions)
- fonctions qui opérent sur des collections de données complexes ou custom

Révéler la complexité par les tests

Fonction à responsabilité unique permet un test unitaire simple.

La difficulté à rediger des tests met en évidence une complexité trop forte et du code inmaintenable. Notamment :

- des stubs enormes
- des mêmes mocks qui gèrent un grand panel de cas
- coupling et difficulté (ou absence compléte) pour injecter les dépendances

Risque d'ignorer des branches ou de désynchroniser les tests de l'implémentation.

Identifier la complexité via outils et intégrations

Des outils d'analyse statique pour la plupart des langages.

Intégration à la CI de la métrique : simple avertissement, refus de merge, refus de push,...

Linters pour intégration dans les IDE (règle `"complexity"` de ESLint, `gocyclo` ou `gocognit`, ...).

Métrique imparfaite, discutable et à utiliser intelligemment

Métrique basée sur la théorie des graphes avec formule mathématique définie strictement.

Il faut du recul quant à notre implémentation dans la réalité.

Les effets de bord et les subtilités de language ne peuvent pas être prise en compte.

Améliorer le score n'est pas un fin en soit : **tradeoffs** à faire selon les **cas métier** ou les besoins d'**optimisation**.