Refactoring : du code qui marche, c'est bien, mais du code maintenable, c'est mieux

Vincent Guyader 1*

Résumé

"N'importe qui peut écrire du code qu'un ordinateur peut comprendre. Les bons programmeurs écrivent du code que les humains peuvent comprendre." *Martin Fowler. (Fowler (1999))*

Formalisé dans les années 90, le refactoring est défini comme un processus systématique de restructuration du code sans en modifier le comportement externe. Il émerge comme un outil essentiel pour améliorer la lisibilité, la maintenabilité et l'efficacité des programmes informatiques. Cette intervention vise à présenter les techniques et bonnes pratiques de refactoring spécifiquement adaptées à l'environnement R.
Mots-clefs : Refactoring, Bonnes pratiques, Maintenabilité, Développement logiciel.

Développement

Nous commencerons par définir le concept de refactoring et son importance dans le développement logiciel. Ensuite, nous explorerons les motifs courants de mauvaise conception du code R et discuterons de la manière dont le refactoring peut les résoudre. Nous mettrons en lumière les principaux avantages du refactoring, notamment l'amélioration de la clarté du code, la réduction de la duplication, et par conséquent, l'amélioration de sa maintenabilité.

Renommage de Variables

Le renommage de variables est une technique de refactoring essentielle pour rendre le code plus compréhensible et cohérent. En R, il est fréquent de rencontrer des noms de variables peu descriptifs ou mal choisis, ce qui peut rendre le code difficile à suivre. En utilisant des noms de variables significatifs et explicites, nous pouvons améliorer la clarté du code et faciliter sa compréhension par les autres développeurs. Par exemple, en remplaçant des noms de variables génériques comme "x" ou "temp" par des noms descriptifs tels que "revenu_annuel" ou "donnees_utilisateurs", nous rendons le code plus explicite et plus facile à maintenir.

```r
# Avant
f <- function(p) {
  s <- 0
  for (i in p) {
    s <- s + i
  }
  return(s / length(p))
}
```

```r
# Après
calcul_moyenne <- function(numbers) {
  moyenne <- sum(numbers) / length(numbers)
  return(moyenne)
}
```

Extraction de Fonctions

L'extraction de fonctions consiste à regrouper des blocs de code répétitifs ou complexes dans des fonctions distinctes, ce qui permet de réduire la duplication et d'améliorer la modularité du code. En identifiant les sections de code récurrentes dans notre programme, nous pouvons les extraire dans des fonctions réutilisables, ce qui facilite la mise à jour et la maintenance du code. Par exemple, si nous avons un bloc de code qui

*ThinkR 1, vincent@thinkr.fr

effectue un calcul spécifique sur plusieurs jeux de données, nous pouvons le regrouper dans une fonction dédiée et l'appeler chaque fois que nécessaire, au lieu de le répéter à plusieurs endroits dans le code.

```r
# Avant
calculate_and_print_average <- function(a, b) {
  average <- (a + b) / 2
  cat("La moyenne de", a, "et",
      b, "est", average, "\n")
}
```

```r
# Après
calculate_average <- function(a, b) {
  return((a + b) / 2)
}

calculate_and_print_average <- function(a, b) {
  average <- calculate_average(a, b)
  cat("La moyenne de", a, "et",
      b, "est", average, "\n")
}
```

Simplification des Expressions Conditionnelles

Les expressions conditionnelles complexes peuvent rendre le code difficile à comprendre et à déboguer. En simplifiant les expressions conditionnelles, nous pouvons rendre le code plus lisible et réduire le risque d'erreurs. En utilisant des opérateurs logiques clairs et en évitant les constructions excessivement complexes, nous pouvons améliorer la clarté et la concision du code. Les expressions conditionnelles complexes cachent généralement des régles métiers importante qu'il convient d'extraire et d'expliciter.

```r
# Avant
if ( !(jour != "Saturday" & jour != "Sunday") | rtt  ) {
    # Instructions
} else {
    # Instructions
}
```

Deviendra :

```r
# Après
if ( en_vacances(jour,rtt)  ) {
    # Instructions
} else {
    # Instructions
}

# on extrait et explicite la règle metier
en_vacances <- function(jour,rtt) {
  week_end <- c("Saturday", "Sunday")
  jour %in% week_end | rtt
}
```

Ces techniques de refactoring, parmi d'autres qui seront abordées, contribuent à améliorer la qualité et la maintenabilité du code R, en rendant le code plus clair, plus concis et plus facile à gérer pour les développeurs.

Cette intervention présentera des cas d'usage de la "vraie vie" et s'adressera aussi bien aux débutants qu'aux programmeurs expérimentés en R, en offrant des conseils pratiques pour améliorer la qualité et l'efficacité de leur code tout au long du cycle de développement. Elle proposera de faire un détour sur le "golden master" préalable nécessaire à tout refactoring et ouvrira sur le concept de Test Driven Development (TDD).

Références

Fowler, Martin. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional.