



# Quand faut-il documenter ou commenter ?

Hoang La

IUT d'Orsay, Université Paris-Saclay

## CONTENTS

Documenter

README

Licence

Documentation du code

Commenter

Choix d'implémentation

Clarifications

Warning

TODO

Mauvais commentaires

Commentaires cryptiques

Commentaires avec trop d'informations


Commentaires redondants

Commenter au lieu de coder proprement

**Laisser du code commenté**

Commenter au lieu de versionner



Documentez pour les utilisateurs, commentez pour les développeurs 

La documentation explique l'utilisation d'un projet, tandis que les commentaires clarifient son code. Toutefois, la documentation sur les aspects techniques peut être utile pour le maintien et l'évolution du projet. Dans ce cours, nous nous limiterons à la documentation de l'interface publique d'un projet et réserverons les commentaires aux détails d'implémentation.

## Documenter

### README



Premier point de contact 

Le **README** fait partie de la documentation car il fournit des informations essentielles sur le projet, telles que son installation, son utilisation et sa configuration. Il sert de guide d'introduction pour les utilisateurs et les développeurs, facilitant la compréhension et l'accès au projet.

## Contenu d'un README

- Titre et résumé du projet.
- Fonctionnalités principales.
- Documentation complémentaire (lien ou instructions pour générer la documentation s'il y en a).
- Instructions d'installation.
- Instructions d'utilisation.
- Licence.

Optionnel :

- Invitation à contribuer.
- Remerciements.
- Informations de contact.
- Démo.

## Exemples de README

[Une liste d'exemples de beaux READMEs.](#)

## Licence

Exemple de la *Licence MIT*, une licence open-source très permissive :

```
Copyright <YEAR> <COPYRIGHT HOLDER>
```

```
Permission is hereby granted, free of charge, to any
person obtaining a copy of this software and associated
documentation files (the "Software"), to deal in the
Software without restriction, including without
limitation the rights to use, copy, modify, merge,
publish, distribute, sublicense, and/or sell copies of
the Software, and to permit persons to whom the Software
is furnished to do so, subject to the following
conditions:
```

```
The above copyright notice and this permission notice
shall be included in all copies or substantial portions
of the Software.
```

```
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO
THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
```

PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

## Documentation du code

Exemple de documentation en Doxygen (qui est très similaire à Javadoc) :

temperature.h

```
#ifndef TEMPERATURE_H
#define TEMPERATURE_H

/**
 * Represents a temperature in Celsius and Fahrenheit.
 *
 * This class allows you to set a temperature in Celsius, convert it to Fahrenheit,
 * and vice versa. It also provides the ability to get the current temperature in either unit.
 */
class Temperature {
private:
    double mCelsius;

public:
    /**
     * Constructs a new Temperature object.
     *
     * Initializes the temperature in Celsius.
     *
     * @param celsius The initial temperature in Celsius.
     */
    Temperature(double celsius);

    /**
     * Gets the current temperature in Celsius.
     *
     * @return The temperature in Celsius.
     */
    double getCelsius() const;

    /**
     * Sets the temperature in Celsius.
     *
     * This method sets the temperature value directly in Celsius.
     *
     * @param celsius The new temperature in Celsius.
     */
    void setCelsius(double celsius);

    /**
     * Gets the current temperature in Fahrenheit.
     *
     * @return The temperature in Fahrenheit.
     */
    double getFahrenheit() const;

    /**
     * Sets the temperature using a value in Fahrenheit.
     *
     * This method converts the given Fahrenheit value to Celsius and sets it.
     *
     * @param fahrenheit The temperature in Fahrenheit.
     */
    void setFahrenheit(double fahrenheit);

    /**
     * Converts and displays the temperature in both Celsius and Fahrenheit.
     *
     */
}
```

```

    * This method prints the current temperature in both Celsius and Fahrenheit.
    */
void displayTemperature() const;
};

/**
 * @example
 * \code{.cpp}
 * Temperature currentTemperature(25.0); // 25°C
 * currentTemperature.displayTemperature(); // Output: Temperature: 25°C / 77°F
 * currentTemperature.setFahrenheit(100.0);
 * currentTemperature.displayTemperature(); // Output: Temperature: 37.7778°C / 100°F
 * \endcode
 */

#endif

```

La syntaxe Doxygen/Javadoc :

```

/**
 * One sentence describing the class/method.
 *
 * More detailed descriptions here with one or multiple sentences.
 *
 * @param parameterName Parameter description.
 * @return Output description if it exists.
 * @throws ExceptionThrown Exception description if it exists (@exception in Javadoc).
 */

```

Un bloc d'exemple en Doxygen :

```

/**
 * @example
 * \code{.cpp}
 * Example usage code with comments on the output.
 * The @example tag does is not built-in in Javadoc but
 * it supports HTML tags.
 * \endcode
 */

```

@example en Javadoc :

```

/**
 * <pre>
 * Example usage:
 * <code>
 * Example usage code with comments on the output.
 * </code>
 * </pre>
 */

```

## Commenter

### Choix d'implémentation

```

// Insertion sort is chosen for simplicity since the data set is small

```

#### Note

Un commentaire expliquant un choix d'implémentation. lorsqu'il existe plusieurs options

Un commentaire expliquant un choix d'implémentation, lorsque il existe plusieurs options possibles, peut être utile. Même si un autre développeur n'est pas d'accord avec le choix fait, il pourra comprendre le raisonnement sous-jacent et l'améliorer si nécessaire.

## Clarifications

```
double calculateLoanPayment(double principal, double annualRate, int months) {  
    // Formula:  $P = (r * PV) / (1 - (1 + r)^{-n})$   
    //  $r$  = monthly interest rate,  $PV$  = present value (loan amount),  $n$  = number of payments  
    double monthlyRate = annualRate / 12;  
    return (monthlyRate * principal) / (1 - pow(1 + monthlyRate, -months));  
}
```

### Formules ou magic numbers

Le commentaire explique les composants de la formule et son application. Sans ce commentaire, une personne non familière avec la formule pourrait avoir du mal à saisir qu'il s'agit d'une formule mathématique, et non un choix d'implémentation technique.

```
// format matched hh:mm:ss MM dd, yyyy  
regex timeMatcher("\\d*:\\d*:\\d* \\w* \\d*, \\d*");
```

### Expressions régulières

Les expressions régulières, bien que souvent complexes, sont indispensables dans notre code. Il est donc utile d'expliquer celle utilisée dans ce cas.

De façon générale, il faut clarifier tout endroit où l'intention et/ou le contexte du code n'est pas clair.

## Warning

```
// Dear maintainer:  
//  
// Once you are done trying to 'optimize' this routine,  
// and have realized what a terrible mistake that was,  
// please increment the following counter as a warning  
// to the next guy:  
//  
// total_hours_wasted_here = 42
```

### Plus sérieusement,

Parfois, il est utile de prévenir un autre développeur d'une erreur souvent rencontrée (quand on essaye d'améliorer un code par exemple) ou souligner l'importance d'une procédure qui pourrait sembler insignifiante.

```
// the trim is real important. It removes the starting
```

```
// and it can be used anywhere as it removes the leading
// spaces that could cause the item to be recognized
// as another list.
string listItemContent = trim(match.str(3));
```

## TODO

```
// TODO : sorting procedure to be implemented
```

## Mauvais commentaires

Beaucoup de commentaires sont une excuse pour du code de mauvaise qualité. Voici les erreurs à éviter.

## Commentaires cryptiques

La racine carrée inverse rapide.

```
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = * ( long * ) &y; // evil floating point bit level hacking
    i = 0x5f3759df - ( i >> 1 ); // what the fuck?
    y = * ( float * ) &i;
    y = y * ( threehalfs - ( x2 * y * y ) ); // 1st iteration
    // y = y * ( threehalfs - ( x2 * y * y ) ); // 2nd iteration, this can be removed

    return y;
}
```

Une version améliorée :

```
#include <csdtint>

/**
 * Computes an approximation of 1 / sqrt(number) using the Fast Inverse Square Root method.
 *
 * This function is based on the original Quake III Arena algorithm.
 * It uses bit-level manipulation to obtain a rough approximation, followed by
 * one iteration of Newton-Raphson refinement for improved accuracy.
 *
 * Explanation:
 * - https://en.wikipedia.org/wiki/Fast\_inverse\_square\_root
 *
 * @param number The input floating-point number.
 * @return An approximation of 1 / sqrt(number).
 */
float fastInverseSquareRoot(float number) {
    static const uint32_t approximationConstant = 0x5F3759DF;
    static const float newtonRaphsonFactor = 1.5F;

    // Interpret the float as an integer for bitwise operations
    union {
        float approximation; // Holds the estimated 1/sqrt(number)
        uint32_t bitwiseRepresentation; // Stores the bitwise equivalent of the float
    } inverseSqrtData = { .approximation = number };

    // Initial approximation using bitwise trick
    inverseSqrtData.bitwiseRepresentation = approximationConstant - (inverseSqrtData.bitwiseRepresentation >> 1);

    // One iteration of Newton-Raphson refinement
    inverseSqrtData.approximation = inverseSqrtData.approximation * newtonRaphsonFactor;

    return inverseSqrtData.approximation;
}
```

```
// One iteration of Newton-Raphson refinement
// A second iteration can improve precision at the cost of time
inverseSqrtData.approximation *= newtonRaphsonFactor - (0.5F * number * inverseSqrtData.approximation);

return inverseSqrtData.approximation;
}
```

## ❶ Documenter les headers, commenter les codes source en C++

Ici, la documentation est intégrée dans le code source afin de garder l'exemple concis. En C++, nous documentons uniquement les headers qui définissent l'interface publique du projet, tandis que les commentaires dans le code source sont réservés aux développeurs.

Dans des langages comme Java, qui ne font pas cette distinction, vous trouverez à la fois de la documentation et des commentaires dans le même fichier.

## Commentaires avec trop d'informations

```
/*
RFC 2045 - Multipurpose Internet Mail Extensions (MIME)
Part One: Format of Internet Message Bodies
section 6.8. Base64 Content-Transfer-Encoding
The encoding process represents 24-bit groups of input bits as output
strings of 4 encoded characters. Proceeding from left to right, a
24-bit input group is formed by concatenating 3 8-bit input groups.
These 24 bits are then treated as 4 concatenated 6-bit groups, each
of which is translated into a single digit in the base64 alphabet.
When encoding a bit stream via the base64 encoding, the bit stream
must be presumed to be ordered with the most-significant-bit first.
That is, the first bit in the stream will be the high-order bit in
the first 8-bit byte, and the eighth bit will be the low-order bit in
the first 8-bit byte, and so on.
*/
```

## Commentaires redondants



**Programmers Memes**  
@ProgrammersMeme

...

Very useful !







Credits: [Programmers Humor](#).

## Commenter au lieu de coder proprement

```
bool isValidPassword(const string& password) {  
    // Pattern matches that:  
    // - Must be between 8 and 20 digits  
    // - Must contain at least one uppercase letter and one lowercase letter  
    // - Must contain one number  
    // - Must not include whitespace  
    // - Must contain one of the following special characters: ! # $ % ^ & *  
    regex pattern("(?=[A-Z])(?=[a-z])(?=[0-9])(?=.*[!#$%^&*])[\S]{8,20}$");  
    return regex_match(password, pattern);  
}
```

### ⚠ Un bon commentaire ?

Le commentaire ci-dessus peut sembler utile puisqu'il explique l'expression régulière difficile à lire utilisée pour valider un mot de passe.

Cependant, ces commentaires ne font que masquer un mauvais code. En suivant les principes du code propre, la fonction `isValidPassword` devrait être refactorisée comme suit.

```
bool lengthInRangeInclusive(const string& str) {  
    return str.length() >= 8 && str.length() <= 20;  
}  
  
bool containsUppercaseLetter(const string& str) {  
    return regex_match(str, regex("(?=[A-Z]).*"));  
}  
  
bool containsLowercaseLetter(const string& str) {  
    return regex_match(str, regex("(?=[a-z]).*"));  
}  
  
bool containsNumericDigit(const string& str) {  
    return regex_match(str, regex("(?=[0-9]).*"));  
}  
  
bool doesNotContainWhitespace(const string& str) {  
    return !regex_match(str, regex("(?=.*[\\s]).*"));  
}  
  
bool containsSpecialCharacter(const string& str) {  
    return regex_match(str, regex("(?=.*[!#$%^&*\\.]"));  
}
```



```
bool isPasswordValid(const string& password) {
    return lengthIsInRangeInclusive(password) &&
           containsUppercaseLetter(password) &&
           containsLowercaseLetter(password) &&
           containsNumericDigit(password) &&
           doesNotContainWhitespace(password) &&
           containsSpecialCharacter(password);
}
```

## Laisser du code commenté

```
InputStreamResponse response = new InputStreamResponse();
response.setBody(formatter.getResultStream(), formatter.getByteCount());
//InputStream resultsStream = formatter.getResultStream();
//StreamReader reader = new StreamReader(resultsStream);
//response.setContent(reader.read(formatter.getByteCount()));
```

### ! Danger

Il arrive souvent de commenter des lignes de code pendant le développement. Cependant, ces commentaires ne doivent pas rester sans explication. Si un autre programmeur lit ce code sans comprendre l'objectif des lignes commentées, il n'osera probablement pas les supprimer. Ainsi, elles risquent de demeurer dans le code, sans justification, et d'induire les futurs développeurs en erreur.

Avec les outils de versionnage comme Git, il est presque impossible de perdre du code (si vous effectuez des commits réguliers !). N'ayez donc pas peur de supprimer du code et de le retrouver plus tard dans l'historique des commits.

## Commenter au lieu de versionner

```
/*
11-Oct-2001 : Re-organised the class and moved it to new package com.jrefinery.date (DG);

05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate class (DG);

05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
*/
```

### ! Danger

Certains pourraient penser qu'il est utile de laisser des notes sur l'évolution du code dans les commentaires pour les autres développeurs.

En effet, cela pourrait être une bonne pratique **si Git n'existait pas**. Avec le versionnage, un simple `git log` suffit (à condition bien sûr que les messages de commits soient clairs) !