NODE Technical Book Club

A Philosophy of Software Design - John Ousterhout

Why Write Comments?

The Four Excuses

1. Good code is self-documenting
2. I don't have time to write comments
3. Comments get out of date and become misleading
4. All the comments I've seen are worthless

Benefits of Well-written Comments

- Main idea behind comments is to capture information that was in the mind of the designer but couldn't be represented in the code.
- Comments will allow others to work on the code more effectively.
- Good documentation reduce cognitive load and unknown unknowns.

A Different Opinion: Clean Code

- In the book Clean Code, Robert C. Martin takes a more negative view of comments.

> “... comments are, at best, a necessary evil. If our programming languages were expressive enough, or if we had the talent to subtly wield those languages to express our intent, we would not need comments very much -- perhaps not at all.”

- **BUT**: Well-written comments are fundamental in defining abstractions and managing complexity.

Comments Should Desribe Unobvious Things

- This should be the guiding principle.
- Unobvious things:
  - Low-level details
  - Design decisions
  - Rules to follow
  - Abstractions

Pick Conventions

- Conventions ensure consistency and easier to write and read.
- Every class should have an interface comment.
- Every class variable should have a comment.
- Every method should have an interface comment.

🚩 Red Flag: Comment Repeats Code

If the information in a comment is already obvious from the code next to the comment, then the comment isn’t helpful. One example of this is when the comment uses the same words that make up the name of the thing it is describing.

Don't Repeat the Code

```
// Obtain a normalized resource name from REQ.
private static String[] getNormalizedResourceNames ( HTTPRequest req)
//  Downcast PARAMETER to TYPE.
private static Object downCastParameter(String parameter, String type)
//  The horizontal padding of each line in the text.
private static final int textHorizontalPadding = 4;
```

No useful information here.

```
// The amount of blank space to leave on the left and
// right sides of each line of text, in pixels.
private static final int textHorizontalPadding = 4;
```

Much better.

Lower-level Comments Add Precision

- Comments can fill in missing details such as:
  - What are the units for this variable?
  - Are the boundary conditions inclusive or exclusive?
  - What does a null value imply?
  - Who is responsible for freeing or closing a resource?
  - Are there certain properties that are always true for the variable?

Bad

```
// Current offset in resp Buffer
uint32_t offset;
```

Better

```
// Position in this buffer of the first object that hasn't
// been returned to the client.
uint32_t offset;
```

Bad

```
// Contains all line-widths inside the document and
// number of appearances.
private TreeMap<Integer, Integer> lineWidths;
```

Better

```
// Holds statistics about line lengths of the form <length, count>
// where length is the number of characters in a line Cincluding
// the newline), and count is the number of lines with
// exactly that many characters. If there are no lines with
// a particular length, then there is no entry for that length.
private TreeMap<Integer, Integer> numLineswWithLength;
```

Focus on what the variable represents, not what it does.

Bad:

```
/* FOLLOWER VARIABLE: indicator variable that allows the Receiver and the
* PeriodicTasks thread to communicate about whether a heartbeat has been
* received within the follower's election timeout window.
* Toggled to TRUE when a valid heartbeat is received.
* Toggled to FALSE when the election timeout window is reset. */
private boolean receivedValidHeartbeat;
```

Better:

```
/* True means that a heartbeat has been received since the last time
* the election timer was reset. Used for communication between the
* Receiver and PeriodicTasks threads. */
private boolean receivedValidHeartbeat ;
```

Higher-level Comments Enhance Intuition

- Higher-level comments should describe the overall intent and structure of the code.
- Commonly used for comments inside methods and for interface comments.

Bad:

```
// If there is a LOADING readRpc using the same session
// as PKHash pointed to by assignPos, and the last PKHash
// in that readRPC is smaller than current assigning
// PKHash, then we put assigning PKHash into that readRPC.
int readActiveRpcId = RPC_ID_NOT_ASSIGNED;
for Cint i = 0; i < NUM_READ_RPC; i++) {
if (session == readRpc[i].session
      && readRpc[i].status == LOADING
      && readRpc[i].maxPos < assignPos
      && readRpc[i].numHashes < MAX_PKHASHES_PERRPC) {
    readActiveRpcld = 1;
    break;
  }
}
```

Better:

```
// Try to append the current key hash onto an existing
// RPC to the desired server that hasn't been sent yet.
```

Higher level comments are harder to write.
Ask yourself:

- What is this code trying to do?
- What is the simplest thing you can say that explains everything in the code?
- What is the most important thing about this code?

Good example:

```
if (CnumProcessedPKHashes < readRpc[i].numHashes) {
  // Some of the key hashes couldn't be looked up in
  // this request (either because they aren't stored
  // on the server, the server crashed, or there
  // wasn't enough space in the response message).
  // Mark the unprocessed hashes so they will get
  // reassigned to new RPCs.
  for (size_t p = removePos; p < insertPos; p++) {
    if (CactiveRpcId[p] == i) {
      if (CnumProcessedPKHashes > 0) {
        numProcessedPKHashes-- ;
      } else {
        if (p < assignPos)
          assignPos = p;
        activeRpcId[p] = RPC_ID_NOT_ASSIGNED;
      }
    }
  }
}
```

Interface Documentation

- The only way to describe an abstraction is through comments.
- If you want code that presents good abstractions, you must document those abstractions with comments.
- If interface comments must also describe the implementation, then the class or method is shallow.

- Interface comments include higher-level information for abstraction and lower-level details for precision:
  - Describe the behavior of the method.
  - Describe each argument and the return value.
  - Describe any side effects.
  - Describe any exceptions.
  - Describe any preconditions.

Example:

```
/**
* Copy a range of bytes from a buffer to an external location.
* \param offset
*       Index within the buffer of the first byte to copy.
* \param length
*       Number of bytes to copy.
* \param dest
*       Where to copy the bytes: must have room for at least
*       length bytes.

* \return The return value is the actual number of bytes copied,
*       which may be less than length if the requested range of
*       bytes extends past the end of the buffer. 0 is returned
*       if there is no overlap between the requested range and
*       the actual buffer.
*/
uint32_t
Buffer: :copyCuint32_t offset, uint32_t length, void* dest)
```

🚩 Red Flag: Implementation Documentation Contaminates Interface

This red flag occurs when interface documentation, such as that for a method, describes implementation details that aren’t needed in order to use the thing being documented.

Bad:

```
/**
* Check if the next object is RESULT_READY. This function is
* implemented in a DCFT module, each execution of isReady() tries
* to make small progress, and getNext() invokes isReady() in a
* while loop, until isReady() returns true.
*
* isReady() is implemented in a rule-based approach. We check
* different rules by following a particular order, and perform
* certain actions if some rule is satisfied.
*
* \return
*   True means the next Object is available. Otherwise,
*    return false.
*/
bool IndexLookup::isReady() { ... }
```

Better:

```
/*
* Indicates whether an indexed read has made enough progress for
* getNext to return immediately without blocking. In addition, this
* method does most of the real work for indexed reads, so it must
* be invoked Ceither directly, or indirectly by calling getNext) in
* order for the indexed read to make progress.
*
* \return
*      True means that the next invocation of getNext will not
*      block (at least one object is available to return,
*      or the end of the lookup has been reached);
*      false means getNext may block.
*/
bool IndexLookup::isReady() { ... }
```

Implementation Comments

- They should describe what and why, not how.
- May not be needed for short methods.
- Longer methods should have a comment before each major block.

```
// Phase 1: Scan active RPCs to see if any have completed.
```

```
// Each iteration of the following loop extracts one request from
// the request message, increments the corresponding object, and
// appends a response to the response message.
```

Cross-module Design Decisions

- Some design decisions unevitably affect multiple modules.
- It is hard to find a good place to document these decisions.
- **Author's suggestion:** Create a central file called designNotes. Then reference it from other modules.

Choosing Names

- Good names are a form a documentation.
- When choosing a name, the goal is to create an image in the reader's mind about the nature of the thing being named.
- Ask yourself: Does the name evoke the right image even in the isolation?

Names Should Be Precise

- The most common problem with names is that they are too generic or too vague.

```
/**
* Returns the total number of indexlets this object is managing.
*/
int IndexletManager::getCount() {...}
```

numActiveIndexlets would be better.

Bad:

```
// Blink state: true when cursor visible.
private boolean blinkStatus = true;
```

Better:

```
// Controls cursor blinking: true means the cursor is visible,
// false means the cursor is not displayed.
private boolean cursorVisible = true;
```

🚩 Red Flag: Hard to Pick a Name

If it’s hard to find a simple name for a variable or method that creates a clear image of the underlying object, that’s a hint that the underlying object may not have a clean design.

Use Names Consistently

- Consistent naming reduces cognitive load.
- Three requirements:
  - Always use the common name for the given purpose.
  - Never use the common name for anything other than the given purpose.
  - Make sure that the purpose is narrow enough that all variables with the name have the same behavior.

Avoid Extra Words

- Words that don't add information just add clutter.
- Common mistakes:
  - Adding a generic noun like "object" or "data".
  - Adding a type name to the variable name.
  - Repeating the class name in the variable name.

A different opinion: Go Style Guide

- The Go programming language argues that names should be very short, often only a single character.
- They claim: "Long names obscure what the code does."

```
func RuneCount(b []byte) int {
  i, n:=0, 0
  for i < len(b) {
    if b[i] < RuneSelf {
      i++
    } else {
      _, size := DecodeRune(b[i:])
        i += size
    }
    n++
  }
  return n
}
```

Do you think this is readable or would you prefer longer names?

Write The Comments First

- The best time to write comments is at the beginning of the process, as you write the code.
- Writing comments first makes documentation part of the design process.
- It results in both better documentation and design.

Delayed Comments Are Bad Comments

- It is common to delay writing comments until the end.
- Consequences:
  - Comments are less likely to be written.
  - Comments are less likely to be useful.

Write The Comments First

- Write the class interface comment first.
- Write interface comments and signatures for the most important public methods.
- Iterate over these comments until the basic structure feels right.
- Write declarations and comments for the most important class instance variables.
- Fill in the bodies of the methods, adding implementation comments as needed.

Benefits:

- Produces better comments.
- Improves the system design.
- Makes comment-writing more fun.

🚩 Red Flag: Hard To Describe

The comment that describes a method or variable should be simple and yet complete. If you find it difficult to write such a comment, that’s an indicator that there may be a problem with the design of the thing you are describing.

Comments