Code Presentation Practice - A New Approach to Source Code Documentation

Yang Cui
G28165607
Software Engineering
Spring 2020

I. Introduction

Documentation has remained a big problem for software engineering throughout the years. Although there are proposals, guidelines and standards for documentation, such as ISO 9001 or IEEE Standard, they are usually too bulky and costly, especially when it comes to source code level. For many projects, source code documentation is just ignored. For those that do documentation, it is usually wasted because either nobody looks the documentation or it is outdated and does not provide enough information on the source code.

This paper tries to propose a light-weighted source code documentation method called the Code Presentation Practice(CPP). The big idea behind CPP is to view the process as not how to document source code thoroughly, but how to present the source code in a way that is friendly to human brains. Thus the name is called code presentation. Another highlight of CPP is that it is largely based on researches and conclusions from cognitive psychology and only uses human nature to enhance the understanding process.

Before jumping into detailed explanations, it is always a good idea to show the goals we are aiming to achieve. The criteria of CPP is as follows.

1) Concise. Here concise does not mean cramp all information into one tiny space. Rather it means only showing the information that is needed and can aid to understanding. In fact too much information will distract the readers.
2) Insightful. The documentation should convey something that cannot be drawn from source code directly. If it can, that is redundant because we can simply read the source code to get the information.
3) Less costly. It should take as little efforts as possible to write and maintain the documentation. Besides, the method should take as little learning time as possible. Ideally developers do not need to learn this method at all. They see and understand by instinct.
4) Consistent. The format and ways of documentation should remain the same across different modules. Whatever the situation, you all document in the same way.
5) General. The method can be applied to most cases, regardless of what programming languages is used, what nature the project is, what scale the project is, etc.

Looking at a counter example will enhance our understanding of what CPP is trying to achieve. This is a detailed UML class diagram from book *Using UML: Software Engineering with Objects and Components*.

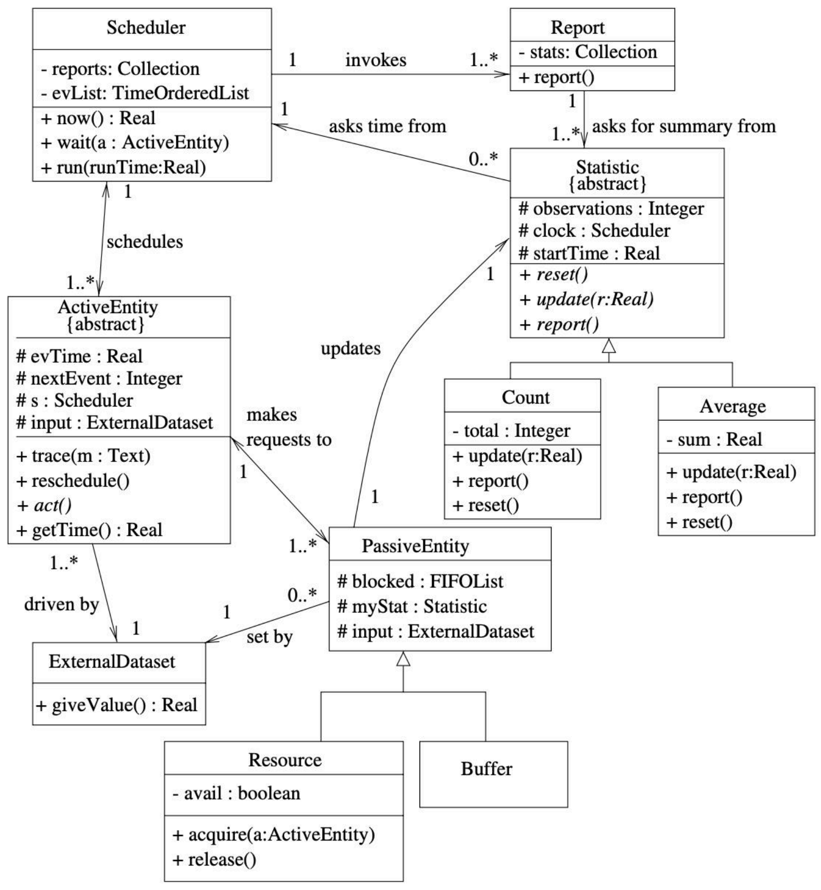


Figure 1 Detailed class diagram for a simulation experiment[1]

The goal here is not to let you look into this diagram, but demonstrate that a detailed class diagram violates the major goals of CPP. First it is not concise. Remember concise does not mean showing as much information as possible in a small space. This diagram is too complex and contain too much irrelevant information. In fact when you first see this diagram, you will have a hard time finding a focus. Second it is not insightful because for all these information you can get from source code. Although you need some time to link between several source codes, it is not a hard job. Third, it surely costs very much to draw this diagram, even with the help of UML tools. Besides, as it shows some detailed information of source codes, like private functions and variables, it needs to be updated each time a single source code changes. And

[1] Perdita Stevens and Rob Pooley, *Using UML : Software Engineering with Objects and Components*, 2nd ed (Harlow, England: Addison-Wesley, 2006), 190.

source code is prone to change. It is very unlikely this diagram will be kept updated during development. Lastly, consistent and general are satisfied. However, satisfying these two without the most important three will do no good.

In short, only people already familiar with the system know how to read this diagram. But this diagram is designed to help new developers understand the system. The detailed class diagram fails its goal.

It may seem impossible to satisfy the 5 goals at the same time. For example if we want it to be general enough, we need to introduce some notations and likely increase the learning cost. But in fact we can, with a little inspiration from cognitive psychology.

II. How Chunks Can Help

Developers have already known the fact that "There is a limit to how much a human can understand at any one time"[2]. Psychologists have looked deeper into this matter and arrived some interesting conclusions.

A. Working memory and chunks

Suppose you are given this task: read a list of words and try to memorize them. You have only 30 seconds to do so. The list is as follows.

Quick, DFS, Merge, 3DES, SHA256, BFS, Heap, Radix, MST, RSA, Bubble, AES

If you actually tried to do this, you will find it very hard to memorize them all. But if the list just contains the first three words, it will be a piece of cake. Psychologists use the term working memory to explain this phenomenon. Working memory provides temporary storage and manipulation of the information necessary for complex cognitive tasks[3]. To evaluate the capacity of working memory, a term chunk is invented. In psychology there is still discussion about what chunk really means. But this is not a psychology paper and for simplicity chunk here just means a unit to store information in working memory. The meaning of unit can vary according to context. For example, to be able to complete the above task, you need to have 12 chunks to remember 12 words. How many chunks do a person have on average? One popular belief is seven. More detailed research suggests that in a verbal, immediate serial recall task, the capacity

[2] Perdita Stevens and Rob Pooley, *Using UML : Software Engineering with Objects and Components*, 2nd ed (Harlow, England: Addison-Wesley, 2006), 6.

[3] Alan Baddeley, "Working Memory," *Science* 255.5044, (Jan 1992): 556.

is limited to three or four[4]. The point is the number is not big and can definitely be counted with fingers.

Another interesting fact about chunks is that we can memorize more by merging smaller chunks into bigger ones. This already happened during the above task. You are memorizing "Quick" as one word(one chunk), not Q-u-i-c-k, five characters(five chunks). In fact if you are from the computer science field, you should do the task a little better because the 12 words are not arbitrarily chosen. They are 12 algorithms from three fields: sorting algorithm, graph algorithm and crypto algorithm. To demonstrate the point clearly as well as for later discussion, here is a graph showing this structure.

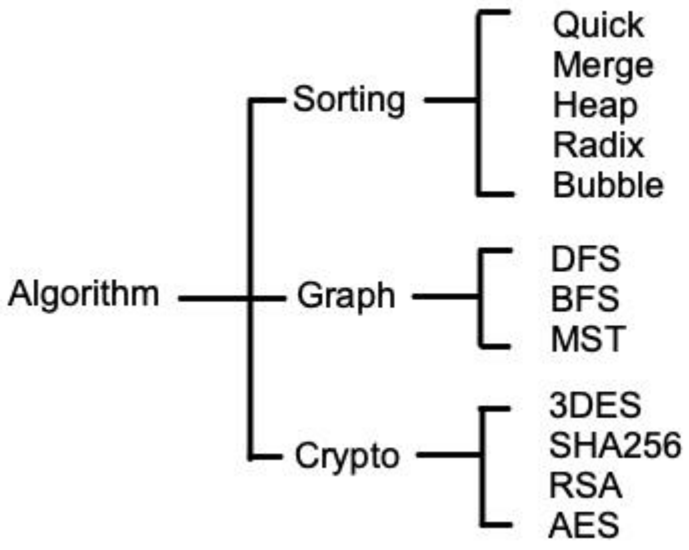


Figure 2 Memorizing Algorithm Names

Chunks in this context are in fact categories that help to organize the words. They, however, can become different things in different context. We will use the term chunk throughout this paper.

B. The Left-Right path and Right-Left path

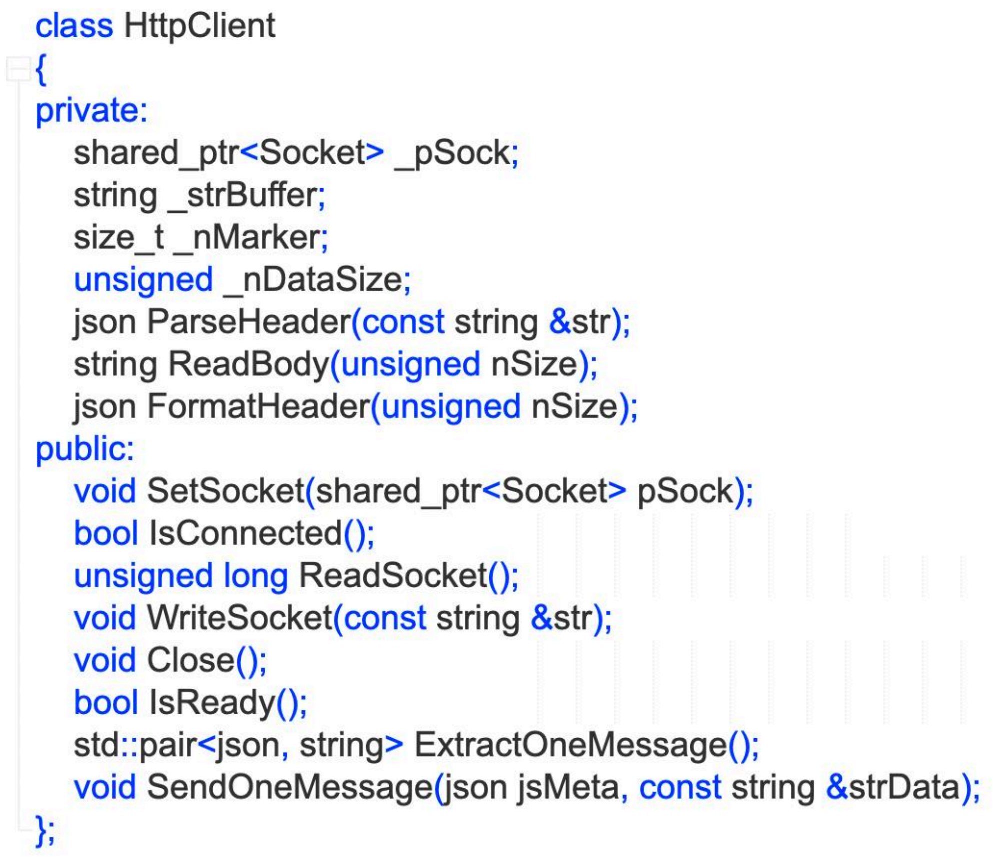
What does chunk has to do with source code documentation? We are not letting developers memorizing source codes anyway. The answer lies in the process of merging chunks. It can be observed that something remarkable happens during the merging of chunks. Why can computer science students merge the chunks while ordinary people cannot? That is because you cannot get the categories just by looking at the words. The words themselves tell you nothing. To

[4] Nelson Cowan, *Working Memory Capacity* (New York: Psychology Press, 2005), 101.

merge the chunks, you need to relate your background knowledge to the current task in a meaningful way. This process is in essence understanding.

The claim is that merging chunks is closely related to understanding. If someone can merge smaller chunks into bigger chunks, we say that person understands the task. I name it the Right-Left path. From the other direction, If we present the bigger chunks first, it can help someone without proper knowledge to understand the task in a significant way. I name it the Left-Right path. Going the Right-Left path is hard because you need to actively search your knowledge base and keep trying to make meaningful connections to current tasks. Going the Left-Right path is easy because you are already pointed out the important concepts and only need to follow them in a natural way. Note that this is not a psychology paper so the terms are used in a loose way.

Let's see how this applies to coding. Here is a piece of header code in C++. What can you make of it?



```
class HttpClient
{
private:
    shared_ptr<Socket> _pSock;
    string _strBuffer;
    size_t _nMarker;
    unsigned _nDataSize;
    json ParseHeader(const string &str);
    string ReadBody(unsigned nSize);
    json FormatHeader(unsigned nSize);
public:
    void SetSocket(shared_ptr<Socket> pSock);
    bool IsConnected();
    unsigned long ReadSocket();
    void WriteSocket(const string &str);
    void Close();
    bool IsReady();
    std::pair<json, string> ExtractOneMessage();
    void SendOneMessage(json jsMeta, const string &strData);
};
```

Figure 3 Typical C++ header

For readers who have no coding experience, this is a mess. For readers who have coding experience but not in C++, at least they can make out class name, member variables and methods. Even for readers who know C++, they still need to read this one by one and try to figure out, or guess what this code is doing.

The mental state can be expressed in the following visual way.

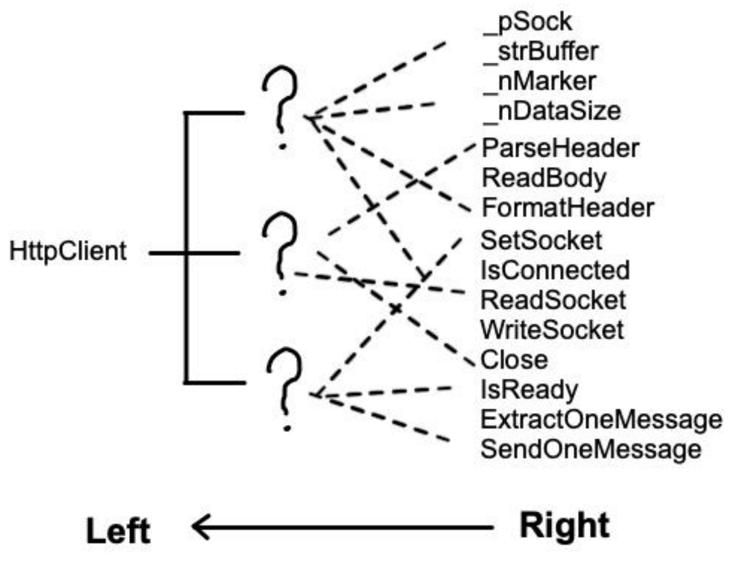


Figure 4 Right-Left path

There are too many small chunks, so you try to understand the code by merging chunks. But it is not clear how to merge them. The only way is to try different combinations. The names can suggest something but not too much. If you happen to know http, it will be easier but it still needs a lot of work.

Now let the author, which is me, to explain how the code is developed.

I have a socket[5] class which can do raw IO, and I want to develop a wrapper to do IO according to http format. The first thing is to provide a send method. Inside the method there should be a format header option, and write socket in raw data stream. Then receive request. That's hard because reading socket can block. I don't want it to block. So I need to first read what's available and store it in a buffer. Then provide a method to check if the buffer is ready to be extracted according to http format. If ready I can extract. Inside extract method I need to read

[5] For readers who are not familiar with network programming. Socket is like a portal that connects a remote computer, or more precisely a particular process on that remote computer. You can read from and write to socket as if you are talking directly to that remote computer. To make the communication meaningful, the two computers should talk using the same format or they won't understand each other. Http is a standard format that helps communication.

the header and body. Finally a method to close the socket. Everything done. Later When I check the code I find checking the buffer needs to parse header each time. That's no good. I can add a marker to indicate how long the body should be. In this way it will run faster.

The above paragraph seems like ramble but shows two important things about coding. First the original authors have some bigger views about the code. They know what the code should do in a high level. Second, Codes are prone to change and grow in an unpredictable way. The authors are likely to add, remove and change any part of the codes.

When the original authors are doing the coding, their mental state can be expressed in this way.

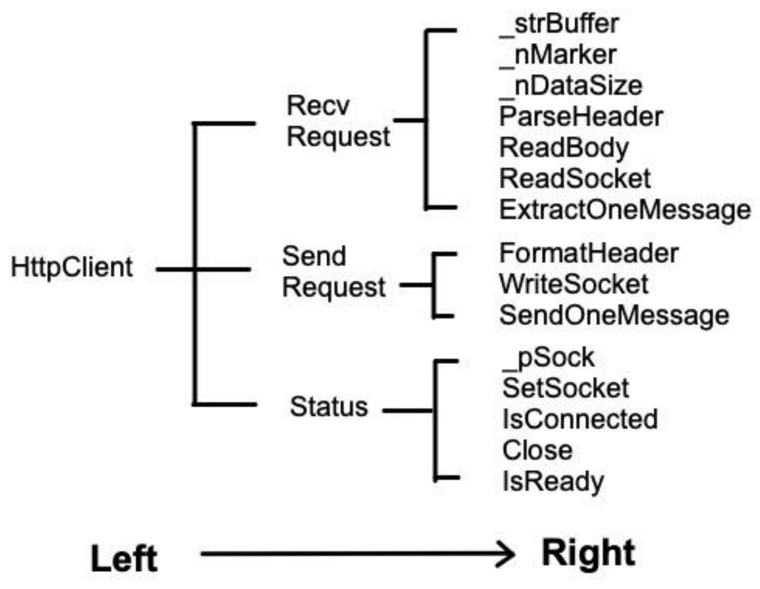


Figure 5 Left-Right path

From their perspective, the code is well-organized because they know the bigger picture, the bigger chunks, the implicit knowledge. As they write and change the code, the text becomes unordered but they still see the code in organized way. When they leave the position the implicit knowledge is lost. So when newcomers see the code, what they see is just unordered codes and sink into figure 4.

This actually explains why source code is so hard to understand in the most general way. The original authors are going the Left-Right path, following the nature flow of thoughts, like differentiating, while the readers are going the Right-Left path, doing reverse engineering all the time, like anti-differentiating. Anti-differentiating is super hard. You need flash of creativity to find the answer. And it is very common no answer can be found.

Now that we have solved the puzzle, it remains to be seen how to overcome this. Again psychology can help.

C. Show the Bigger Chunks Explicitly

We need to show the bigger chunks explicitly, but how? Gestalt can shed light on this matter. Gestalt Psychology is an early school in psychology that have done great contributions to image perception. They have purposed many concepts and principles but we will use only two here. Figure/ground states that when we focus on an object, the object becomes the figure and all other things fade into ground. We will only see the figure, ignoring the ground. Principles of Similarity states that if the shapes of individual objects are similar, we tend to notice the pattern and group them together. A typical example would be similarity between the elements in alternate rows cause the row percept to dominate[6].

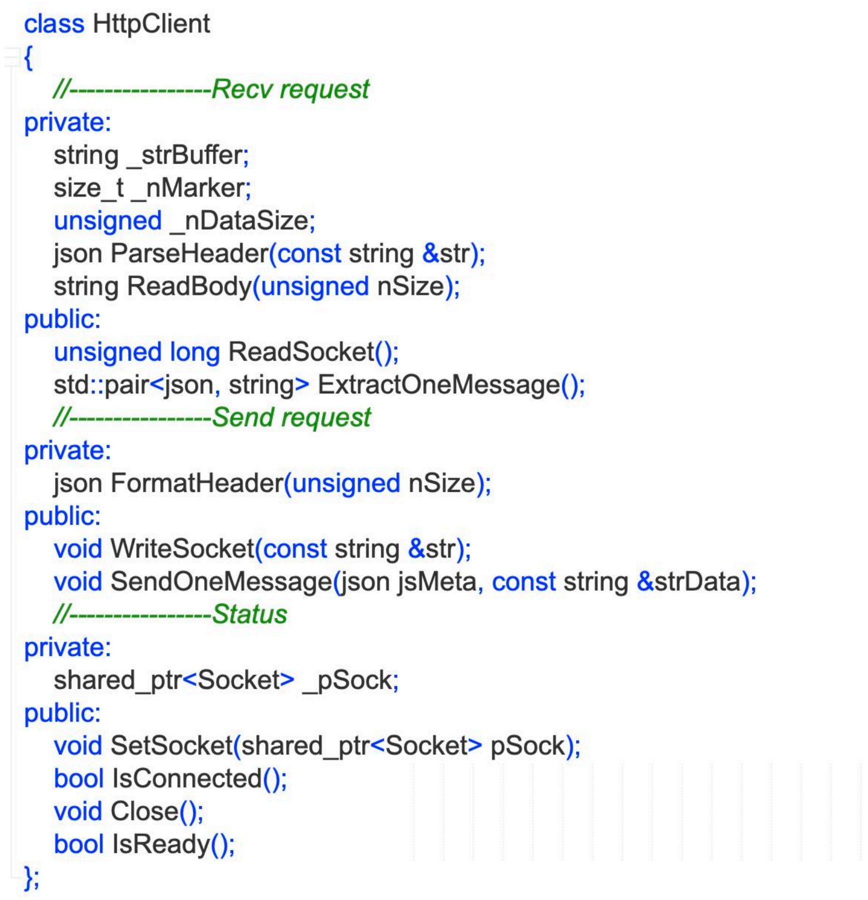
Based on these conclusions, the first practice of CPP would be:

CPP #1: For declaration, use comment lines to separate symbols into different chunks, with a brief description following each line.

This is simple. Note that the line should be long enough to visually form a horizontal separator. I suggest using 16 hyphens. Also you should not add other comments that occupy a single line since that will act like distractions. Do the comments after the symbol in the same line

With CPP #1, the earlier messy C++ header can be formatted like this.

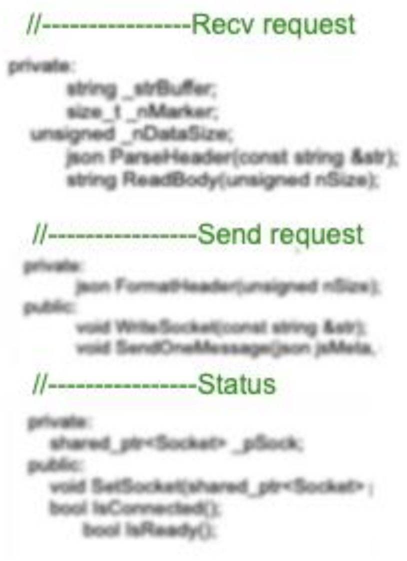
[6] Colin Ware, Information Visualization : Perception for Design (San Francisco, CA: Morgan Kaufman, 2004), 190



```
class HttpClient
{
   //----------------Recv request
private:
   string _strBuffer;
   size_t _nMarker;
   unsigned _nDataSize;
   json ParseHeader(const string &str);
   string ReadBody(unsigned nSize);
public:
   unsigned long ReadSocket();
   std::pair<json, string> ExtractOneMessage();
   //----------------Send request
private:
   json FormatHeader(unsigned nSize);
public:
   void WriteSocket(const string &str);
   void SendOneMessage(json jsMeta, const string &strData);
   //----------------Status
private:
   shared_ptr<Socket> _pSock;
public:
   void SetSocket(shared_ptr<Socket> pSock);
   bool IsConnected();
   void Close();
   bool IsReady();
};
```

Figure 6 C++ header formatted by CPP #1

Although in the compiler's eye the codes are the same, it makes a huge different in the human's eye. Notice what catches your attention when you first look at the figure. According to Gestalt principles, it would be something like this.



```
//----------------Recv request
private:
        string _strBuffer;
        size_t _nMarker;
   unsigned _nDataSize;
        json ParseHeader(const string &str);
        string ReadBody(unsigned nSize);

//----------------Send request
private:
        json FormatHeader(unsigned nSize);
public:
        void WriteSocket(const string &str);
        void SendOneMessage(json jsMeta,

//----------------Status
private:
    shared_ptr<Socket> _pSock;
public:
    void SetSocket(shared_ptr<Socket> ;
    bool IsConnected();
        bool IsReady();
```

Figure 7 Gestalt principles applied to CPP #1

In this way the readers are forced to notice the chunks, which aid to form the Left-Right path. As a result it can provide firm ground for understanding rest of the codes.

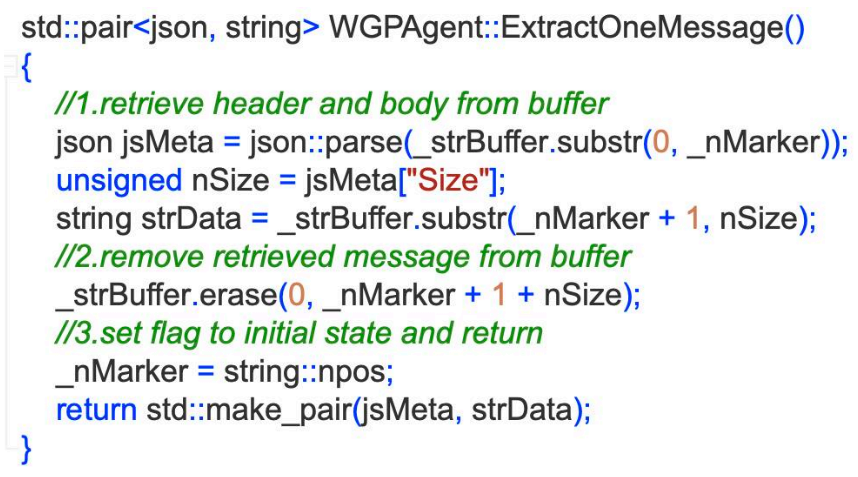
Furthermore, this method can in some way help to check the design of classes. Remember the average chunks an ordinary person have is around five. We will just take that at face level. This means that if a class can be separated into more than 5 chunks, it is an indication of bad design. Maybe the class is taking too much responsibilities.

This is just the header file. We should do the same in implementation as well.

CPP #2: For implementation, use numbered comment to separate steps into different chunks, with a brief description of the goal following the number.

The implementation part is always a few lines of codes executed in sequential order to do something. That means we can always group the codes into certain steps. By explicitly showing what goal each step is trying to achieve, we are making the implementation much clearer to the readers. Note that we do not use comment line here because different developers have different styles for comment. We should keep CPP minimal and save space for individual styles.

An example of the implementation could be like this.



```
std::pair<json, string> WGPAgent::ExtractOneMessage()
{
    //1.retrieve header and body from buffer
    json jsMeta = json::parse(_strBuffer.substr(0, _nMarker));
    unsigned nSize = jsMeta["Size"];
    string strData = _strBuffer.substr(_nMarker + 1, nSize);
    //2.remove retrieved message from buffer
    _strBuffer.erase(0, _nMarker + 1 + nSize);
    //3.set flag to initial state and return
    _nMarker = string::npos;
    return std::make_pair(jsMeta, strData);
}
```

Figure 8 C++ implementation formatted by CPP #2

When readers see this piece of code, they can develop an overall concept of what the codes are doing without looking into the actual codes. This gives them the implicit knowledge about the codes. After that they can do whatever they want depending on their aims.

D. Evaluation

Do CPP #1 and #2 meet the criteria?

1) Concise. Just a few lines of comment are added to the codes and they are essential to understanding.

2) Insightful. The added comments convey the implicit knowledge of the codes. These information cannot be drawn from the text easily.

3) Less costly. The comments cost almost nothing to write because when doing the coding the authors already organized the codes in chunks. They just need to type a few words, nothing compared to writing long documentations and drawing graphs. Since there is no external documentation there is no need to update other files. The maintenance cost happens when the codes are changed, but right on the spot. The methods just exploit human instinct and do not need to be learned. Even readers who do not know CPP can understand the codes easily in this format.

4) Consistent. Only one format for declarations and one for implementations. In fact the #1 and #2 are the same thing. Just a slight variation to keep things simple.

5) General. As long as it is the code, it can be chunked. The methods only use comment to do the trick and every programming language supports comment. Depending on the language there may not be declarations, but still codes can be chunked like this. Fold all implementations and the class will look like header file. In fact I suggest readers do the folding first when viewing codes so that the implementations will not act as distractions.

Using CPP #1 and #2 can boost understanding, but we can still do better to present the code.

III. How hierarchy Can Help

So far we have chunked the codes into big categories. It can be observed that inside each chunk the symbols are still in a chaotic state. This leaves room for improvement.

A. Hierarchical structure

Hierarchy is everywhere. It is so common that it becomes our instinct. We usually use a tree structure to express hierarchy and this applies to all cases. A typical example would be organizational structure inside a company.

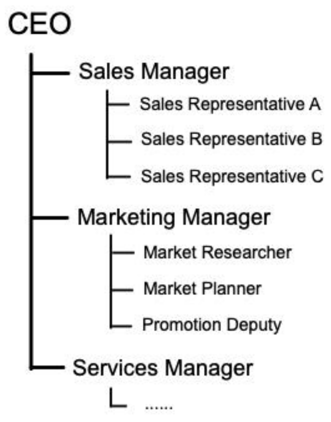


Figure 9 Hierarchical organization inside a company

Now I want to point out the similarity and differences between hierarchy and hierarchical chunks.

In a sense they are interchangeable. Recall the algorithm memorizing task. 12 words are too many so we merge smaller chunks into bigger chunks to help us remember the words. What we get after merging chunks is in fact a hierarchy structure. Psychologists have shown that hierarchical chunking can greatly improve our memory capacity[7]. You can also view the organization structure of a company as bigger chunks containing smaller chunks. This can help you memorize the structure.

[7] Nelson Cowan, *Working Memory Capacity* (New York: Psychology Press, 2005), 81.

But there is something fundamentally different. Hierarchy conveys more information than hierarchical chunks. The information is about the relationships among items. Look at the company organization. When sales manager is placed under CEO, we are not saying the manager is a CEO, very different from bubble sort is a sorting algorithm. What we are expressing is CEO is the superior and sales manager is the inferior. As a result CEO can tell sales manager to do this and that but not vice verse. This logic also goes the same for sales manager and sales representatives.

By formatting the hierarchical structure we are in fact putting the whole system together, from lowest level to highest level, in an organized way. And we do not need to specific the relationship between each item. We just use the dominating paradigm and apply to all relationships. This will simplify the system and thus help us understand it in a quick way.

This relationship is what we will take advantage of for the source code documentation task. In short we will be organizing the symbols inside a chunk into hierarchical structure. It will give us more understanding about source codes other than information provided by chunks. However, the dominating paradigm is a little off the target in the given context. We will interpret the relationship as “has something to do with”, or under most cases, “uses”. This will become clearer in the next part.

B. Organize in a hierarchical way

Since the tree model makes sense to all people, we will directly use that and format the text layout into a tree-like structure.

CPP #3 Inside each chunk, use indents to format symbols into tree structure.

There are some points need clarifying.

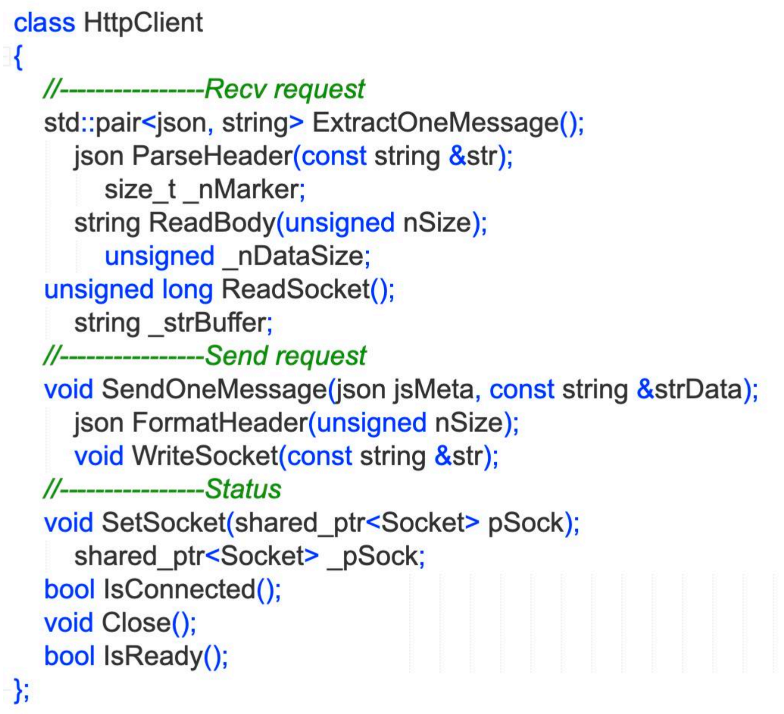
First, what if a symbol is used across chunks? This can happen frequently. The answer is to put it where it makes most sense to the author. Depending on the author’s understanding, format could be different.

Second, do not consider access control at first because public, private and protected keywords will act like distractions. Ideally simple symbols can be used like +, -, #. But most languages do not support this so just adding the decorator at last, in front of each symbol. We do not consider access control because these things do not help to construct the hierarchical structure, which means do not aid understanding.

Third, although the relationship means “uses”, do not take that at face level. Put symbols under symbols which form a logical “uses”, not a text “uses”. That is if a method simply uses a symbol in its implementation but not really logically connected, do not place them together.

Fourth, by convention the most important symbol, usually the API, is placed at the top inside each chunk.

Let’s look at the example of HttpClient code.



```
class HttpClient
{
    //----------------Recv request
    std::pair<json, string> ExtractOneMessage();
        json ParseHeader(const string &str);
            size_t _nMarker;
        string ReadBody(unsigned nSize);
            unsigned _nDataSize;
    unsigned long ReadSocket();
        string _strBuffer;
    //----------------Send request
    void SendOneMessage(json jsMeta, const string &strData);
        json FormatHeader(unsigned nSize);
        void WriteSocket(const string &str);
    //----------------Status
    void SetSocket(shared_ptr<Socket> pSock);
        shared_ptr<Socket> _pSock;
    bool IsConnected();
    void Close();
    bool IsReady();
};
```

Figure 10 C++ header formatted by CPP #3

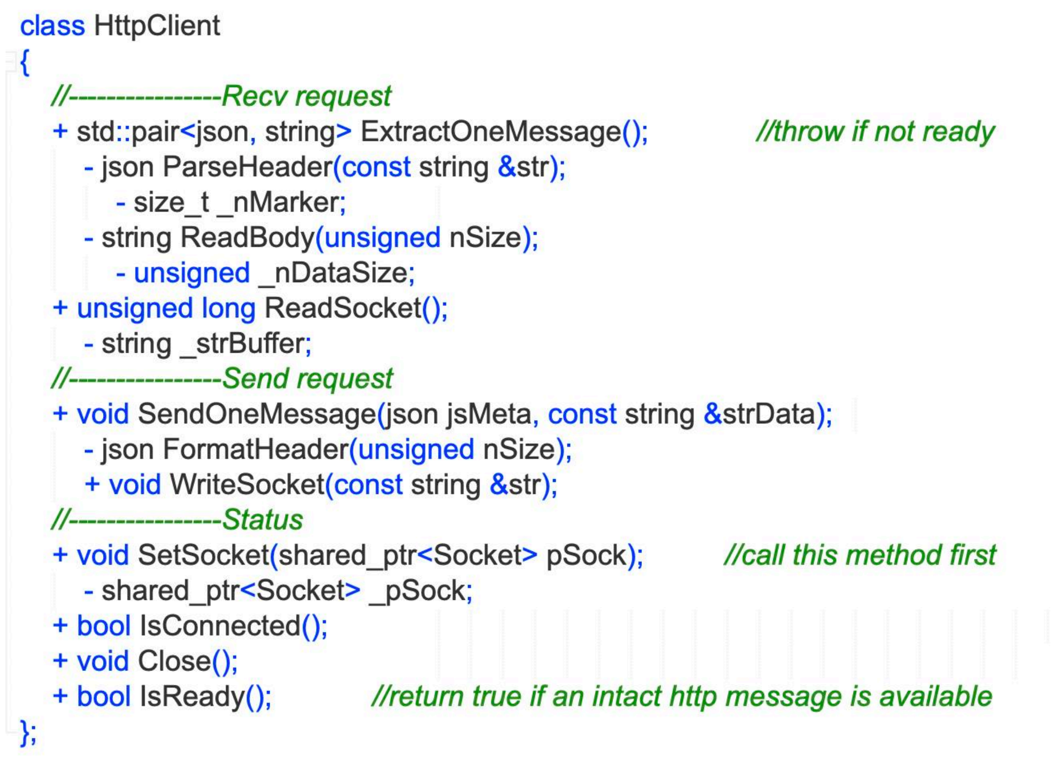
The first question you should ask is “Does this make a difference?”. The tree structure is not obvious. The first thing you noticed should still be the chunks. That is exactly how it is intended to work. The tree structure should provide information only after you decide to read more into the details. It should not manifest itself at first glance. Remember too much information is a bad thing. Second we shall see how the tree can help our understanding. The access control decorators are omitted for clarity.

Inside the Recv chunk, the first one is ExtractOneMessage(). By convention this should be the most important method in this chunk. That is to say we use this method to perform the receive http function. The next layer is ParseHeader() and ReadBody(). This suggests that the ExtractOneMessage() will use ParseHeader() and ReadBody() to perform the actual task. As the names suggest, ParseHeader() will get the header of http request and ReadBody() will produce the body of http request. After getting header and body ExtractOneMessage() will return them. Now a layer deeper. Inside ParseHeader() there is the variable _nMarker. From the name we can guess ParseHeader() uses a marker to determine where is the header. We do not know how exactly it is determined, but the marker probably plays a big part in this. Inside ReadBody() _nDataSize is used. This suggests that _nDataSize could be the body size. It is used to determined how much data to read. Now we go back to the top layer and see ReadSocket(). There is a _strBuffer inside. It probably means this method reads raw data from socket and store them into a string buffer. The next chunk is Send. There is only one top layer method SendOneMessage(). This means when we want to send http request, we only need to call this method. Inside it there are FormatHeader() and WriteSocket(). This method probably uses FormatHeader() to generate http header and then WriteSocket() to actually send header and body outside. The Status chunk is rather simple so we will skip it.

After reading the tree, what can you get? You should feel amazing because we somehow grasped the major implementation details without looking at implementations at all. And we are not far from truth. I already told you how I wrote the code so you can confirm this. This is the power of CPP #3. Of course there is an art on how to interpret the tree, but it is not hard to master and you already learned it.

Some additional explanations. Clearly ReadSocket() and WriteSocket() both use socket, which is variable _pSock. But I placed it under SetSocket() because I think SetSocket() determines the variable. Placing it here makes the most sense. Similarly buffer is used in both ParseHeader() and ReadBody(). But ReadSocket() actually fills the buffer so this is the determine one. Where to place the symbols reflects how authors see the code.

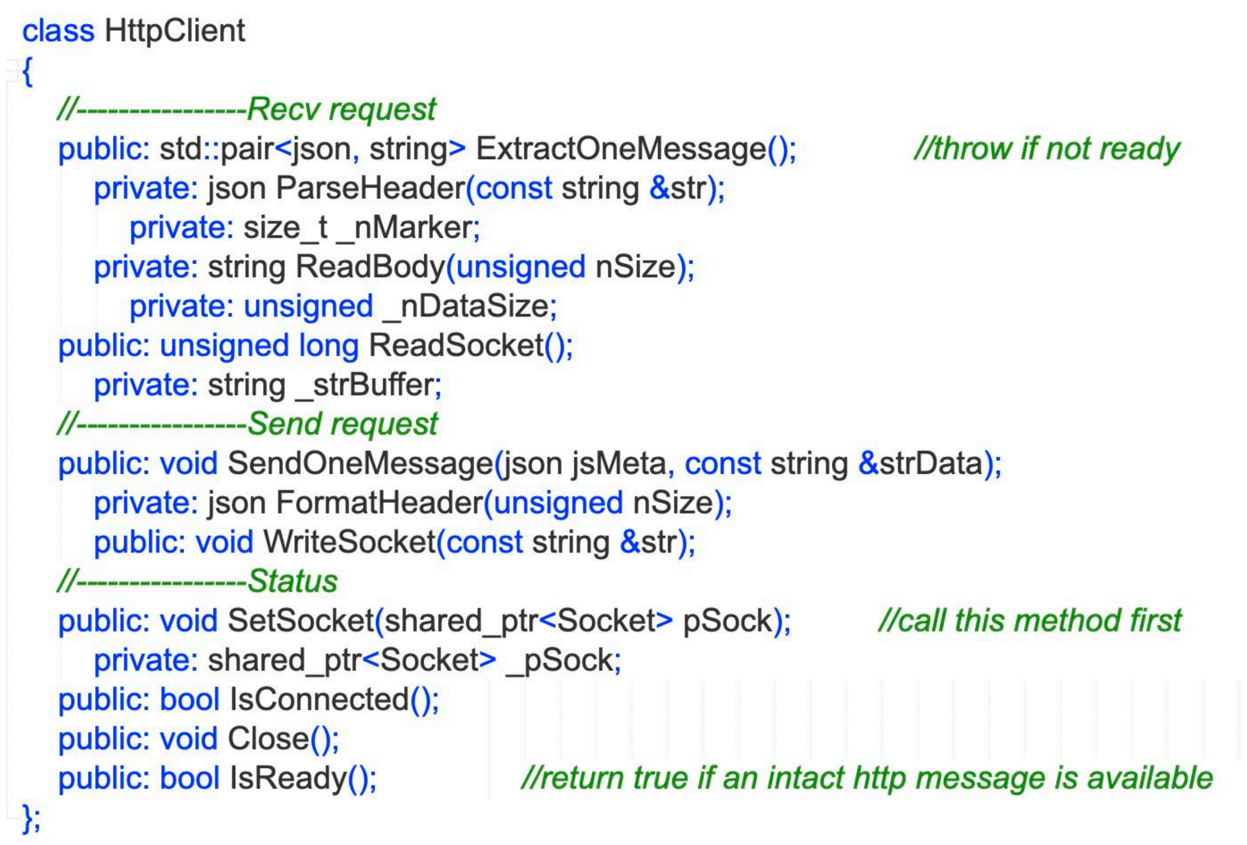
Finally we should put access control decorators to the codes. Ideally the code could be something like this.



```
class HttpClient
{
   //----------------Recv request
   + std::pair<json, string> ExtractOneMessage();            //throw if not ready
      - json ParseHeader(const string &str);
         - size_t _nMarker;
      - string ReadBody(unsigned nSize);
         - unsigned _nDataSize;
   + unsigned long ReadSocket();
      - string _strBuffer;
   //----------------Send request
   + void SendOneMessage(json jsMeta, const string &strData);
      - json FormatHeader(unsigned nSize);
      + void WriteSocket(const string &str);
   //----------------Status
   + void SetSocket(shared_ptr<Socket> pSock);          //call this method first
      - shared_ptr<Socket> _pSock;
   + bool IsConnected();
   + void Close();
   + bool IsReady();          //return true if an intact http message is available
};
```

Figure 11 Ideal code formatted by CPP #3

Using +,-,# only occupies one character and will keep distraction at minimal. However, this feature is not supported by most language. As far as I know only Object-C supports this. To make the code compilable, replace the signs with ordinary decorators. This will give codes like this.



```
class HttpClient
{
   //----------------Recv request
   public: std::pair<json, string> ExtractOneMessage();          //throw if not ready
      private: json ParseHeader(const string &str);
         private: size_t _nMarker;
      private: string ReadBody(unsigned nSize);
         private: unsigned _nDataSize;
   public: unsigned long ReadSocket();
      private: string _strBuffer;
   //----------------Send request
   public: void SendOneMessage(json jsMeta, const string &strData);
      private: json FormatHeader(unsigned nSize);
      public: void WriteSocket(const string &str);
   //----------------Status
   public: void SetSocket(shared_ptr<Socket> pSock);        //call this method first
      private: shared_ptr<Socket> _pSock;
   public: bool IsConnected();
   public: void Close();
   public: bool IsReady();            //return true if an intact http message is available
};
```

Figure 12 Compilable code formatted by CPP #3

As you can see the decorators somewhat distract the readers. But if the readers get used to it, like Java developers, this should not be a big problem. But I still say there is beauty inside the format in figure 11.

C. Evaluation

Does CPP #3 meet the criteria?

1) Concise. Just a few indents and we can understand the major implementation details. We cannot expect more.

2) Insightful.The tree structure conveys meanings that cannot be drawn from text. Of course you can read the implementation but it takes much time and efforts to do so.

3) Less costly. The indents and structure cost almost nothing because when doing the coding the authors already have the structure in mind. They just need to format a bit, nothing compared to writing long documentations and drawing graphs. Since there is no external documentation there is no need to update other files. The maintenance cost happens when the codes are changed, but right on the spot. The tree structure is commonsense but to interpret it correctly you need to learn the art. However it is fairly easy compared with learning big stuffs like UML.

4) Consistent. The tree structure is the only thing that needed to be formatted.
5) General. This needs some discussion. To format the tree only indents are used. Most programming languages can cope with it but one notable exception is Python, which uses indentation as code block. For these few languages this practice will not work. For languages that do not separate declarations and implementations, the code can still for formatted according to CPP #3, though it may look a bit strange. But once the implementations are hidden from view the code will look the same as in figure 12.

CPP #1, #2 and #3 are all manipulating the source code text directly. There is, however, a limit how source code can convey. For the next part we will be putting the final touch outside source code.

IV. Documentation from the User's Eye

A. Two types of description

Suppose you are asked to describe an object, say a bike. How will you describe it?

One way is like this. A bike is composed of frame, steer, wheels, brakes, etc. It is typically made of steel. It weighs XXX and can load a weight of YYY. It was invented in 19th century. Some famous bike manufacturers are ZZZ.

Another way is like this. Bike is mainly used as a way of transportation. Riding a bike saves energy and moves much faster than on foot. Besides, bikes can be used to carry heavy goods. Some types of bike, like sport bikes, are designed for recreations and competition.

Although both descriptions increase our understanding about bikes, they differ in a fundamentally way. The first one I call scientific approach, which tries to describe an object as it is in an objective manner. The second one I call utilitarian approach, which tries to describe how an object can be used in a subjective manner. The word utilitarian used here does not strictly refer to Utilitarianism, which is kind of normative ethics in philosophy[8]. They are, however, related. Utilitarianism judges a thing based on what goodness or happiness it can bring. If something can be used to achieve some goals for the user, it will normally bring happiness to the user. Thus describing an object in the perspective of usage somehow fits in with Utilitarianism thinking. That is the reason I choose the word utilitarian here. It is different from the scientific view, where the happiness of users is not a concern.

---

[8] Julia Driver, "The History of Utilitarianism", *The Stanford Encyclopedia of Philosophy* (Winter 2014 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/win2014/entries/utilitarianism-history/>.

There is one interesting attribute for the utilitarian approach. The number of descriptions is limited, usually one or two, once a specific user is given. For example, if you are in a hurry and want to go from place A to place B, all you care about a bike is that it moves faster than on foot. This nice attribute will save a lot of troubles for source code documentation. There are just two types of people that will look at the source code: those who want to use the code and those who need to maintain the code. Certainly nobody looks at the code to find plagiarism or do proofreading for comments. For the first kind, they do not care how the code is written. They just want to know how the class can be used to achieve certain goals. For the second kind, they want to know the structure of the code, which is already demonstrated through CPP #1, #2 and #3, and how this class interact with other modules in the system. This indicates we do not need to write a lot in the document. As long as we address the needs for these two users we are good.

Another claim is both the scientific approach and the utilitarian approach are necessary for a complete understanding of object. Lacking one will result in a partial understanding. I will not prove this claim because we are approaching this matter in a utilitarian way and this feels like commonsense. So far CPP #1, #2 and #3 are providing scientific view about the source code. We will be adding the utilitarian view so that the documentation becomes complete.

B. Add Utilitarian description

Now that we have identified two kind of readers, we should design a format that can answer their questions and be used in all situations. A unified format will save the efforts of thinking what to write as well as promote understanding among developers. It may seem strange but a very inspiring source would be Rampolla's book on writing in history. He wrote that "Historians come to their work with a deep curiosity about the past; to satisfy that curiosity, they ask some of the same questions detectives ask: Who? What? When? Where? And Why?"[9]. It is very interesting to think source code as historical facts and new readers as historians. Indeed, the original author is gone, leaving only the source code buried in the sand of time. The newcomer somehow recovered the code, only to find it shrouded in mystery. He then asks himself several questions: Who wrote this code? What is he trying to achieve? When is the code finished? Where is the code used? Why did he write like this? And usually the newcomer cares this question: How to use this code? Developers cares more about what the code can do to current system than historians do about facts to contemporary world. Based on this line of thought, the fourth practice, though in a bit different flavor, goes like this.

[9] Mary Lynn Rampolla, *A Pocket Guide to Writing in History*, 7th ed (Boston: Bedford/St. Martin's, 2012), 2-3.

CPP #4: write a utilitarian description comment of what, why, who+where, when+how in front of class declaration, with one unit matched with one sentence only.

The what unit aims to describe what the class can do. To help format the language, you can start with "this class is used to solve XXX problem". But do not write this sentence down. That is redundant.

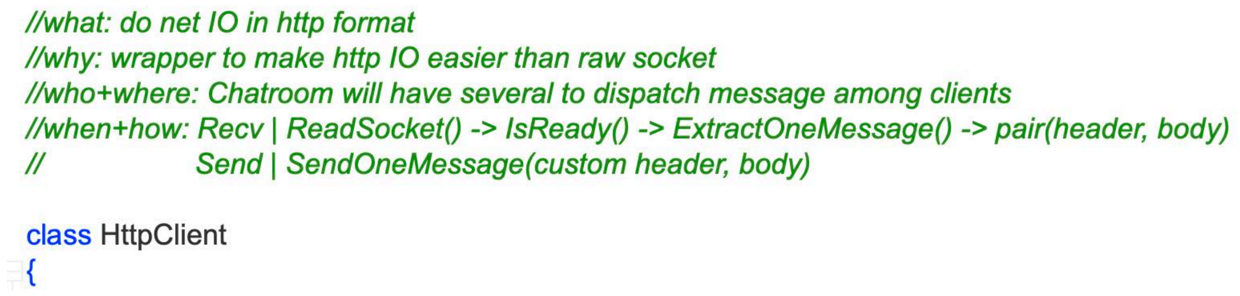
The why unit aims to offer rationale for this class. Rationale can mean a lot. In this case you can start with "This class is needed because …" or "This class is designed like this because …".

The who+where clause can contain several units. It aims to describe the relations with other modules. It mainly states who will use this class or where the class will be used. You can specify certain module if the relationship is significant and stable, say "XXX class will hold one to do XXX task", or in a more general way, say "whenever you need to do XXX, create an instance".

The when+how clause can contain several units. It aims to describe under what situation how specific method shall be used in what way. You should always give an example of how to call the method if it is non-trivial.

Here are some additional notes. First, write only one sentence for each unit. More is not better. If you feel there is a lot to write it may indicate this class is taking too many responsibilities. Second, you can mention detailed symbols from other modules here because this is documentation at source code level. If mentioning detailed names help understanding, do it. Third, keep the syntax simple and do not use a lot of "and, or".

Here is an example of utilitarian description for the http code.



```
//what: do net IO in http format
//why: wrapper to make http IO easier than raw socket
//who+where: Chatroom will have several to dispatch message among clients
//when+how: Recv | ReadSocket() -> IsReady() -> ExtractOneMessage() -> pair(header, body)
//               Send | SendOneMessage(custom header, body)

class HttpClient
{
```

Figure 13 utilitarian description by CPP #4

That is pretty straightforward and makes the readers understand the code better. The what clause tells the readers what to expect for this class. The why clause seems useless at first but if thinking from the other direction you can be sure that this class does not provide vital functions for the whole system. As a result this class is kind of standalone. The who+where clause indicates a specific container and gives the readers a general idea where this class is expected to be found and used. This is good because now the users have more knowledge about the relationship between this class and the whole system. The when+how clause demonstrates how to use this class. If the readers just want to use this class instead of maintaining it, looking at the utilitarian description is enough. The rest of the code can be ignored. A productive practice.

C. Evaluation

Does CPP #4 meet the criteria?

1) Concise. Just a few sentences in a fixed format. It is surely better than long documentation.
2) Insightful. The description provides information that can never be drawn from code. Besides it gives the general idea of the code.
3) Less costly. You still need to write a few sentences. But it is right at the spot and just text. Writing and rewriting costs little compared with doing it in an external documentation. It just uses natural language so everyone can read without problem.
4) Consistent. The only format is the four clauses structure and will always be like this.
5) General. The description just uses comments so all programming languages support it. It remains to be seen whether the four clauses can cover all situations. There is no good way to prove it since it relies on experience and reasoning. Maybe more psychological researches can provide answers to this question.

V. Experiment on the effectiveness of CPP

The four practices are all the contents of CPP. The next question would be does CPP really help readers understand the code better? The good news is we can do psychological experiments to confirm the effectiveness of CPP. The bad news is currently I do not have the resources to actually perform the experiments. This section will just discuss an experiment methods and a concrete example to help readers of this paper understand CPP better.

A. Experiment methods

To get the experiment prepared, we will pick 100 participants who have done some coding in the past. Then we will divide the participants into two groups, A and B. During the division we need to make sure experienced participants and not so experienced participants are mixed evenly. We do this because CPP does not reply on coding experience of the readers. It just uses human instinct to do the trick. However, in the general sense the more experience you have the better you can understand the code. By mixing participants evenly we try to eliminate the experience factor during the experiment. This suggests, unlike many other experiments, the group is not randomly formed. The experience can be judged by the number of years the participants have done in coding. Another factor is the programming language. In the general sense the more familiar you are with certain language the better you can understand the code written in that language. To make sure the participants are in the same ground, we pick one language that all participants know. To research the language aspects of CPP we can carry out the experiments many times in different languages. For now let us assume we pick C++.

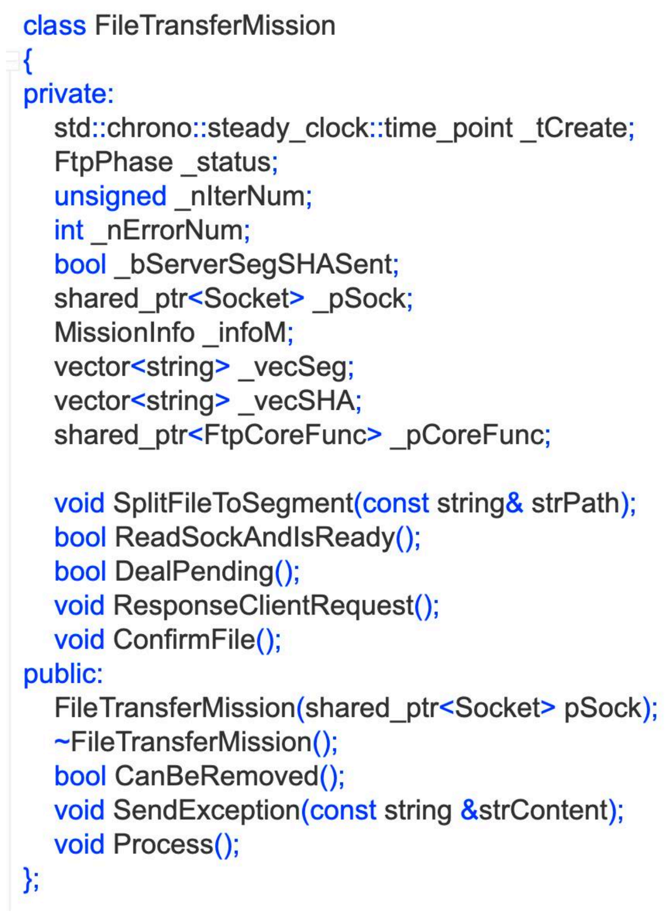
After the groups are formed, we show different code to different groups. Group A will see a typical C++ class header and implementation. They will read the code for 10 minutes. Group B will first see a short article introducing CPP for 3 minutes. Then they will see C++ class header and implementation formatted in CPP for 7 minutes. Here is the reason why the process differs. At first it may seem reasonable to present code formatted in CPP to Group B and let them also read for 10 minutes. If the experiment is carried out like this, it focuses on how human instinct plays a part in CPP, not how effective CPP is compared to typical coding. Although CPP almost costs nothing to learn, the readers still need to grasp its core concept and practice. However, if Group B first learn CPP and then spend 10 minutes read the code, it will be unfair because Group B have extra time and knowledge than Group A. As a result there is a split in time.

Finally all participants are asked to answer some structured questions about the code to test their understanding. Structured questions, rather than open questions, are used here because they are easier to handle and analyze. The questions should test the comprehensive understanding of the code, not only how the code is implemented, but also how to use the code. Each question has some scores. We evaluate which group better understanding the code by the average scores.

B. Test for readers

The following non-trivial code could be a good test to see if readers understand CPP. It can also be used for the experiments above. I will only show the header file here because the essence of CPP lies in header. I highly encourage the readers to take this test here.

This is a typical C++ header. What can you make out in 5 minutes? Hint: do not try very hard because you will not achieve much.



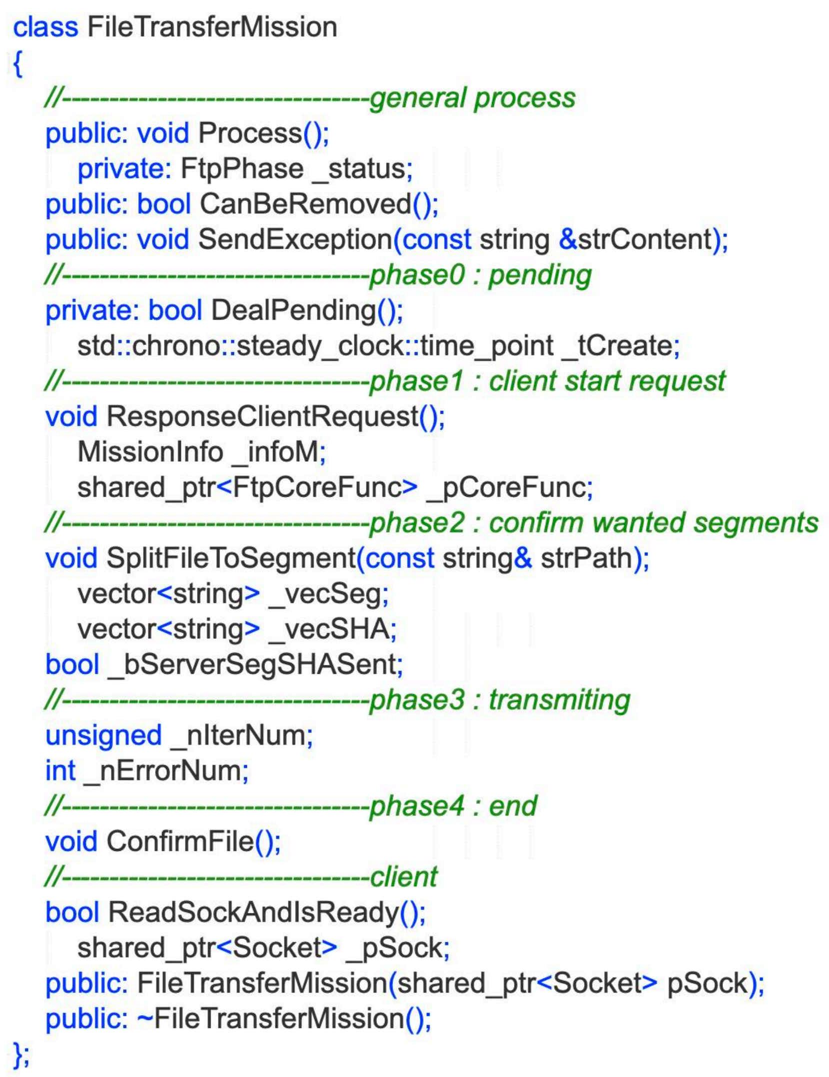
```
class FileTransferMission
{
private:
    std::chrono::steady_clock::time_point _tCreate;
    FtpPhase _status;
    unsigned _nIterNum;
    int _nErrorNum;
    bool _bServerSegSHASent;
    shared_ptr<Socket> _pSock;
    MissionInfo _infoM;
    vector<string> _vecSeg;
    vector<string> _vecSHA;
    shared_ptr<FtpCoreFunc> _pCoreFunc;

    void SplitFileToSegment(const string& strPath);
    bool ReadSockAndIsReady();
    bool DealPending();
    void ResponseClientRequest();
    void ConfirmFile();
public:
    FileTransferMission(shared_ptr<Socket> pSock);
    ~FileTransferMission();
    bool CanBeRemoved();
    void SendException(const string &strContent);
    void Process();
};
```

Figure 14 Test - typical C++ header

In fact you cannot arrive at meaningful conclusions just by looking at the header. All you can know is that this class has something to do with file transmitting through the Internet. This is suggested by the class name. Each piece can give some information but it is unclear how they are related.

Now try the header formatted by CPP. The utilitarian description is not shown here. I suggests the readers to try their best here.



```cpp
class FileTransferMission
{
  //-------------------------------general process
  public: void Process();
    private: FtpPhase _status;
  public: bool CanBeRemoved();
  public: void SendException(const string &strContent);
  //-------------------------------phase0 : pending
  private: bool DealPending();
    std::chrono::steady_clock::time_point _tCreate;
  //-------------------------------phase1 : client start request
  void ResponseClientRequest();
    MissionInfo _infoM;
    shared_ptr<FtpCoreFunc> _pCoreFunc;
  //-------------------------------phase2 : confirm wanted segments
  void SplitFileToSegment(const string& strPath);
    vector<string> _vecSeg;
    vector<string> _vecSHA;
  bool _bServerSegSHASent;
  //-------------------------------phase3 : transmiting
  unsigned _nIterNum;
  int _nErrorNum;
  //-------------------------------phase4 : end
  void ConfirmFile();
  //-------------------------------client
  bool ReadSockAndIsReady();
    shared_ptr<Socket> _pSock;
  public: FileTransferMission(shared_ptr<Socket> pSock);
  public: ~FileTransferMission();
};
```

Figure 15 Test - C++ header formatted by CPP

In fact we can get the major construct of this class just by looking at the header. Here is a detailed explanation.

This first thing you should observe is the chunks are titled with phaseX. This suggests the mission is carried out in phases. This is actually the key part to understanding this class. Now let's examine each chunk.

The top chunk is general process. By convention it should be the most important chunk of this class. The first method is Process(), which indicates it can be the main API. Under it is a _status variable. This means Process() has something to do with status. It either uses status or changes status. Both can happen. Considering there are different phases for the mission, it is very likely that Process() will perform different tasks depending on status and change status when the task is finished. The other two methods in this chunk do not give much information. We can skip them.

The phase0 chunk is called pending. Under DealPending() there is a clock called _tCreate. This suggests that when the mission is created, the creation time is stored and mission starts to wait until something happens. That is all we can make for this phase. Why there is a pending phase we can not tell now.

The phase1 chunk is called client starts request. This could be an answer to chunk0: the mission is waiting for message from client. Under ResponseClientRequest() there are mission info and a FtpCoreFunc. We can guess that after message from client arrives, the class reads request to extract mission information and response to client. It is still not clear what FtpCoreFunc means. As the name suggest, it may provide the core function for file transfer, like uploading or downloading. However, we can tell it is largely determined during phase1. If we have the source code we can definitely look it up there for further information.

The phase2 chunk is called confirm wanted segments. The method is SplitFiletoSegment() and uses _vecSeg and _vecSHA. It probably reads a file, splits it and stores the segments in _vecSeg. Besides there is a _vecSHA so we can guess the method will calculate SHA for each segment and store it in _vecSHA. This also indicates that this class has some error correction mechanism. The meaning of that bool variable is unclear. But we know it is used during phase2.

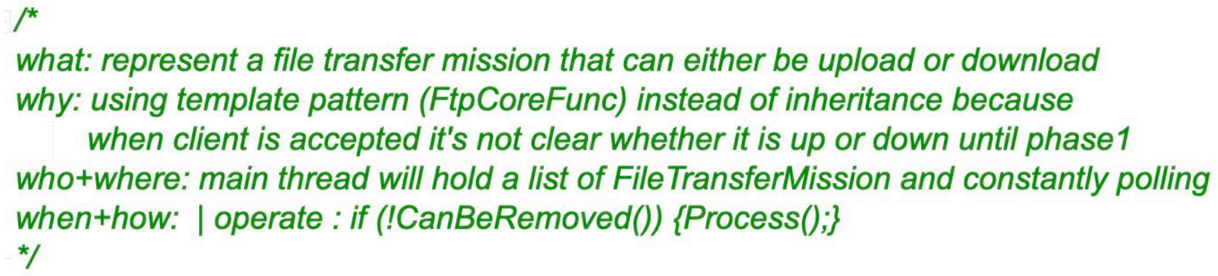
The phase3 chunk is called transmitting. The actual file transmitting should happen in this phase. There is a variable called _nIterNum. Considering the file is already split into segments, we may say the file is transmitted segment by segment in an iterative way, and the variable is used to indicate which segment is being transmitted. The error number variable may play a part in the error correction mechanism.

The last phase is end. The ConfirmFile() method probably is used to check whether the file on client and on server is the same. If they are the same then the mission is completed.

There is also a chunk called client. Inside the chunk the socket appears many times. Thus the socket is a representation of remote client. This can be confirmed by the constructor's input argument: to construct a mission you need to pass in a socket.

Now the interpretation is over. What do you think? It is remarkable that it takes so many paragraphs to describe the header while CPP conveys all the meanings with a few words and indents. Those paragraphs are the traditional documentations and kept in a separate file. No programmers have the motivation to write that. However, using CPP all is different.

Finally let's see the utilitarian description of the code.



```
/*
what: represent a file transfer mission that can either be upload or download
why: using template pattern (FtpCoreFunc) instead of inheritance because
      when client is accepted it's not clear whether it is up or down until phase1
who+where: main thread will hold a list of FileTransferMission and constantly polling
when+how:  | operate : if (!CanBeRemoved()) {Process();}
*/
```

Figure 16 Test - utilitarian description formatted by CPP

With this description all things should be as clear as crystal. The what clause confirmed our understanding about the mission. It can be uploading or downloading. The why clause provides a rationale for the design, especially the previously unclear part of FtpCoreFunc. The who+where clause gives a broader view of the whole system. We now know the general behavior of the server and how this class fits into it. The when+how clause confirmed our understanding about the Process() method. Besides we know how to invoke the class.

I hope this comprehensive example can help the readers better understand CPP.

VI. Conclusion

CPP is a novel method that tries to deal with documentation at source code level. There are four practices of CPP.

CPP #1: For declaration, use comment lines to separate symbols into different chunks, with a brief description following each line
CPP #2: For implementation, use numbered comment to separate steps into different chunks, with a brief description of the goal following the number
CPP #3 Inside each chunk, use indents to format symbols into tree structure
CPP #4: write a utilitarian description comment of what, why, who+where, when+how in front of class declaration, with one unit matched with one sentence only.

These four practices meet the criteria of concise, insightful, less costly, consistent and general. They are sufficient to serve as documentation for source code.

Now the readers should understand why it is called "code presentation practice". It realizes that the text layout of source code has meanings to human eyes and tries to organize the information that is most friendly to human brains. The goal is to present the code to others, even the future you, in a meaningful way.

There is also a big implication from CPP. In traditional view coding and documentation are separate processes, and they somehow hinder each other. However, CPP views coding and documentation as the head and tail of the same coin. On the one hand, coding is doing documentation at the same time, only with a few extra comment lines and indents. On the other hand, thinking how to do documentation can in fact help the developers sort their thoughts and write better code. Furthermore it can save a lot of potential headaches for others and, more importantly, themselves.

This view can, hopefully, solve the problem of motivation. People should do a lot of things but they don't. Developers should do the traditional documentation but they lack the motivation to do it. In CPP coding and documentation help each other. Doing one activity automatically achieves the other, like killing two birds with one stone. As a result, developers have at least more motivations to follow CPP than traditional documentation.

More work remains to be done on the effectiveness of CPP. If resources available, experiments designed in section V can be carried out in a large scale. It will provide empirical evidence for the promotion of CPP.

References

Baddeley, Alan. “Working Memory.” *Science* 255.5044 (Jan 1992): 556–559.
Cowan, Nelson. *Working Memory Capacity*. New York: Psychology Press, 2005.
Driver, Julia, "The History of Utilitarianism", *The Stanford Encyclopedia of Philosophy* (Winter 2014 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/win2014/entries/utilitarianism-history/>.
Fernández-Sáez, Ana et al. “Does the Level of Detail of UML Diagrams Affect the Maintainability of Source Code?: a Family of Experiments.” *Empirical Software Engineering* 21.1 (2016): 212–259.
Land, Susan K., and Walz, John W. *Practical Support for ISO 9001 Software Project Documentation Using IEEE Software Engineering Standards*. Hoboken, N.J: J. Wiley and Sons, 2006.
Martin, M. “Local and global processing: The role of sparsity.” *Memory & Cognition* 7, 476–484 (1979). https://doi.org/10.3758/BF03198264.
Rampolla, Mary Lynn. *A Pocket Guide to Writing in History*. 7th ed. Boston: Bedford/St. Martin’s, 2012.
Scanniello, Giuseppe et al. “Do Software Models Based on the UML Aid in Source-Code Comprehensibility? Aggregating Evidence from 12 Controlled Experiments.” *Empirical Software Engineering* 23.5 (2018): 2695–2733.
Stevens, Perdita., and Pooley, R. J. *Using UML : Software Engineering with Objects and Components*. 2nd ed. Harlow, England: Addison-Wesley, 2006.
Wagemans, Johan et al. “A Century of Gestalt Psychology in Visual Perception: I. Perceptual Grouping and Figure–Ground Organization.” *Psychological Bulletin* 138.6 (2012): 1172–1217.
Wagemans, Johan et al. “A Century of Gestalt Psychology in Visual Perception: II. Conceptual and Theoretical Foundations.” *Psychological Bulletin* 138.6 (2012): 1218–1252.
Ware, Colin. *Information Visualization : Perception for Design*. San Francisco, CA: Morgan Kaufman, 2004.