In this document, the implementation details of the pcynlitx platform has been demonstrated.

The author: ERKAM MURAT BOZKURT, M.Sc in Control Systems Engineering

e-mail: pcynlitx.help@gmail.com

PCYNLITX IMPLEMENTATION DETAILS

1 HOW PCYNLITX DIFFERS CURRENT MULTITHREADING TOOLS

The theoretical foundation of the PRG-MTP Systems has been given in the research article. Basically, PCYNLITX is a typical PRG-MTP System which is developed particularly for multithreded software development and it is a first prototype of its king. In a more strict definition, it can be said that PCYNLITX is a programmable meta programming system which produces cybernetic control libraries for C++ multithread programming applications. In fact, different from the other multithreading tools, PCYNLITX is not a multithreading library. Instead, it is a multithreading library generator and in each library construction process, it produces a new "*project-specific cybernetic control library for multihreading*" based on the programmer needs. In this definition, the term "*project specific cybernetic control library*" refers a multithreading library that is written for the particular requirements of a multithread software project. However, it is obvious that this software development strategy (*producing a new library for each software application*) is completely different from the classical multithreaded software development process. Therefore, at first, the necessity of this new approach must be explained. Indeed, as indicated before, each multithreaded software system has different structure. In this definition, structure reflects the basic elements which may effects the scheduling of the multithreaded software. In order to emphasise the necessity of project specific control system design process, the structure of a multithreaded software and its effect on the multithreaded software development process has been illustrated in below.

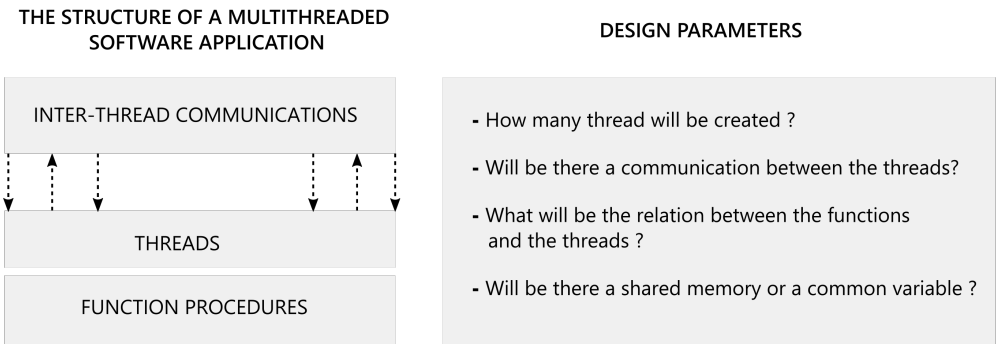


Figure 1: The structure of a multithreaded software

On the figure which is given in above, the design parameters of a multithreaded software are listed and these parameters may change on each multithreaded software development process. For instance, on each multithreaded software, the number of the threads which will be created differs based on the requirements of the software and the computational resources. Unfortunately, the diversity on the parameters which effect the scheduling of the multi-threaded software doesn't allow developing of a unified control system library for multithreading. Because, the control system must control every parameter which may effect the scheduling of the threads. Therefore, in PCYNLITX applications, a new project specific control system is designed in each software development process. In order to achieve this, PCYNLITX platform must known the context of the software to be developed before cybernetic control system design. Thus, in PCYNLITX applications, the structure of the multithread software and the design parameters of the software to be developed are determined before library construction. More specifically, in each software development process which PCYNLITX is used, the programmers must perform declarations about the architecture of the software to be developed to the PCYNLITX. By this way, the PCYNLITX System can learn the context of the software to be controlled and it performs a projection to the requirements of the cybernetic thread control system such as the memory areas to be allocated and their data types. Then, PCYNLITX system constructs the necessary thread management tools in a compact form. Strictly speaking, the project specific library construction process allows the construction of data structures holding thread specific data. In fact, in order to provide cybernetic thread management, intelligent thread control system must collect some thread-specific data such as the ID numbers given by the operating system to the threads or the status of the threads in therms of blocked or non-blocked in runtime.

2 The structure of the PCYNLITX applications

The structure of the PCYNLITX applications includes the hints about how cybernetic thread management system works. A Typical PCYNLITX application consists of an object that governs how function routines are executed by the threads. In the PCYNLITX applications, many tasks such as thread creation, object sharing and thread synchronization are performed by a single object. In fact, this object is a composition of the objects which perform different tasks. In the previous sections, this object has been named as cybernetic control object. In the PCYNLITX applications, this object is named as "*server object*" for easy understanding and the class which defines the type of the server object is named as "*server class*". The elements of a typical PCYNLITX application have been shown in below figure.

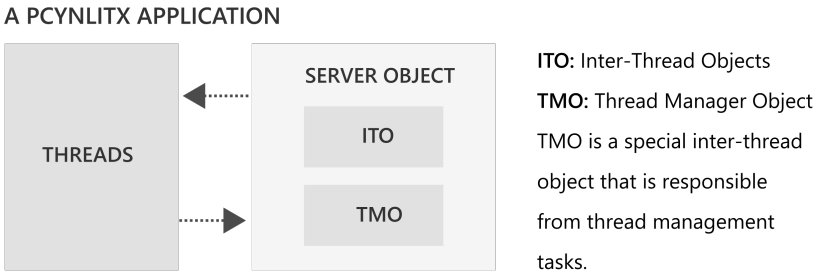


Figure 2: A typical PCYNLITX application

In the PCYNLITX applications, every variable is shared as an object between the threads. Therefore, the server class is designed as an object server including each object which is shared between the threads. For abbreviation, in the following of this paper, the objects which are shared between threads will be named as "*inter-thread objects*" and the types of these objects will be named as "*inter-thread classes*". Basically, the inter-thread objects are used in inter-thread communication and the operations which are performed between the threads. In the figure, the object which is named as "*thread manager object*" is a special inter-thread object which is responsible from the thread synchronization tasks. All of the inter-thread objects including thread manager object are the members of the server object. In every PCYNLITX application, an instance of the server class is created and it is responsible from every tasks related with the management of the process such as thread creation, object sharing between the threads and cybernetic executions of the threads. Thanks to the declarations which are performed before library construction, the server object is constructed with exact information about the threads and their function routines. In other words, before the creation of the threads, the server object exactly knows how many thread will be created on the process and what will be the name of the thread functions to be executed.

3 HOW META PROGRAMS WORK IN PCYNLITX

The PCYNLITX includes many special meta-programs that are used in automatic code generation. On library construction process, at first the necessary information is extracted from the header files specified by the programmer. Then, the PCYNLITX writes the new code files based on the information coming from reading process. In figure-2 (a), the pseudo code illustrating how the header files are read by the PCYNLITX has been given and In figure-2 (b), the pseudo code illustrating how the new code files are written by the PCYNLITX has been given.

```
Open a the target code file
Read the first code line = > "class sample"
If string includes "class",   extract "sample"
Read the next code line = > " {"

If the readed caracter is "}",  end operation.
Close the file.
```

(a)

```
Open a new empty file
Write into file = > "\n  class Thread_Manager"
Write into file = > "\n  {"
Write into file = > "\n  public:"

Write into file = > "\n };"
Close the file.
```

(b)

Figure 3: How meta programs work in Pcynlitx

Basically, the header files are read line by line and the necessary information is extracted from the header files. In the case of code writing, the codes are written based on the code templates which are defined as character lists which are in the form of separated code lines. In the code templates, the places which require application specific information are filled according to the information coming from the other units of the PCYNLITX.

4 THE STRUCTURE OF THE PCYNLITX

Pcynlitx is an open source, free to use software which has been developed as a result of scientific research study and it is distributed with GNU GPLv3 license. In Pcynlitx, before the construction of an application specific multithreading library, the programmers perform declerations about the architecture of the software to be developed to the Pcynlitx. In other words, the programmers perform restrictions on the architecture of the multithreaded programs before coding. By this way, the classes and/or data structures which can hold many thread specific data such as thread ID numbers can be constructed by the Pcynlitx. However, in order to achieve this, Pcynlitx needs to improve its global knowledge about the software to be developed. Therefore, before the library construction process, the Pcynlitx collects information about the context of the program from the programmer by means of a "descriptor file". This descriptor file acts as an interface between the programmer and the Pcynlitx. From a different perspective, descriptor file can be considered as a contract which is signed between the Pcynlitx and the programmer before the library construction process. Now, let we start to investigate the structure of the Pcynlitx and how an application specific library is constructed by the Pcynlitx. In the following of this technical paper, the meta-programs which construct application-specific threading library will be introduced briefly. The basic structure of the Pcynlitx platform is given below figure.

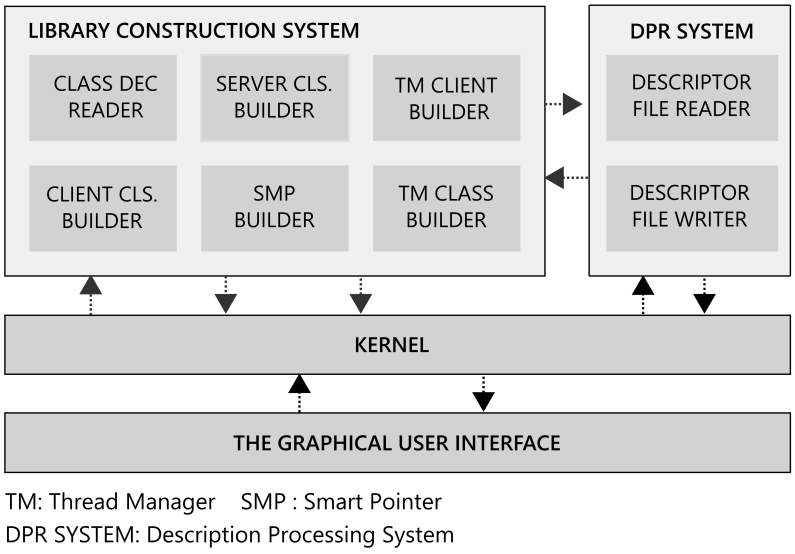


Figure 4: The structure of the pynlitx

4.1 Description Processing System

In fact, in the library construction process, the first task is receiving *the specifications of the multi-threaded application to be developed* from the user of the Pcynlitx. This data receiving process is carried out by a sub-system of Pcynlitx and this sub-system is named as "Description Processing System". Description Processing System consists of two meta program. The first one of them is "Descriptor File Writer" that writes the requirements coming from the user into the descriptor file. Before the library construction, the users enter their preferences about the software to be developed to the graphical user interface of Pcynlitx. At the same time, the descriptor file is coded automatically by "Descriptor File Writer". In the description processing system, the second meta-program is the "Descriptor File Reader" and it reads the descriptor file which is written. In the library construction process, at first, the descriptor file is read line by line by the Descriptor File Reader and the necessary data are received by means of some reading rules determined previously by the Pcynlitx. After the descriptor file is read, each sub-system of Pcynlitx receives its input data from the descriptor file reader.

4.2 Server Class Builder

In the Pcynlitx applications, many tasks such as thread creation, object sharing and thread synchronization are performed by a single object. In fact, this object is a composition of the objects which perform different tasks. In the pcynlitx documentations, this object is named as "*server object*" and the class which defines the type of the server object is named as "*server class*". For abbreviation, in pcynlitx programming, the objects which are shared between threads are named as "*inter-thread object*" and the types of these objects are named as "*inter-thread classes*". The basic structure of the server class has been shown in below figure

```
class Server {
public:
  Member Functions;
  Inter-Thread Objects
  Address Passing Pointers
  Data types for parameter passing
private:
  std:thread Thread[i]

}
```

i: Represents the number of the threads to be created on the process

Inter-Thread Objects: They are shared between the threads

Address Passing pointers: They hold the address of the Inter-Thread Objects

Data types: They holds thread specific data and a pointer pointing address passing pointers

Figure 5: The basic structure of the server class

In Pcynlitx applications, each inter-thread object is a member of the server class. An instance of server class is constructed in main thread. and in Pcynlitx, "Server Class Builder" writes the server class.

4.3 Client Class Builder

In library construction process, Pcynlitx defines some helper classes in order to call the member functions of the inter-thread objects from main thread's stack. The instances of these classes can be used for indirection from the function routines executed by the threads (*the thread functions*) to the main thread. In pcynlitx programming, these classes are named as "*client classes*" and the instances of the client classes are named as the "*client objects*" as well. In the library construction process, the meta-program which is depicted with the name "Client Class Builder" in the figure-2 is responsible from the construction of the client classes.

4.4 Thread Manager Builder

The "Thread Manager Object" is a special type "inter-thread object" which is responsible for the thread synchronization tasks and the "Thread Manager Builder" is the meta-program which writes the thread manager class. "Thread Manager Client Builder" is the meta-program which writes the client of the thread manager class.

4.5 Class Declaration Reader

In the figure-2, the meta-program which is depicted with the name "Class Declaration Reader" is shown as a separate class for easy understanding. But, in fact, each meta-program which is shown in the library construction system has its own class declaration reader. In essence, the meta-program which is depicted with the name "Class Declaration Reader" reads the declarations of the C++ header files and it receives the information which is necessary in the library construction process from the header files.

4.6 Smart Pointer Builder

The meta-program which is depicted as "Smart Pointer Builder" in the figure-2 writes the smart pointers pointing the variable types which are defined by the user. Just as the other classes, these smart pointers are designed in the client-server taxonomy as well. Therefore, the clients of these pointers are also written by the Pcynlitx.

5 INTER THREAD COMMUNICATION ON PCYNLITX APPLICATIONS

In Pcynlitx, anything which is shared between the threads must be shared as in the form of object (*the inter-thread objects*) and for optimum usage of the process memory, only the addresses of these objects are shared between the threads. Therefore, the server class has some pointer members that point its own members (*the pointers pointing to the inter-thread objects*) and with these pointers, the server object passes the addresses of its own members to the newly created threads. However, this parameter passing operations are performed in two stages. In first stage, the addresses of the inter-thread objects are holds by a data structure having pointer members in the types of inter-thread objects. Then, another data structure is used in order to pass both the address of this data structure and some thread specific data. Therefore, before the generation of the server class, two different application specific data structures are defined automatically by the "Pcynlitx" in order to pass the addresses of the inter-thread objects to the threads. The first one of them is the data structure which includes pointer members that point to the inter-thread objects. In the following of this paper, this data structure will be named as "*inter-thread data structure*". This data structure has also a standard pointer member that points the object which is previously named as "*thread manager object*". On the other hand, the second data structure includes an integer member and a pointer member pointing to the previously defined inter-thread data structure. In this paper, this data structure will be named as "*thread specific data structure*" and it is used in order to pass the necessary information to the newly created threads. These data structures are shown in below figure.

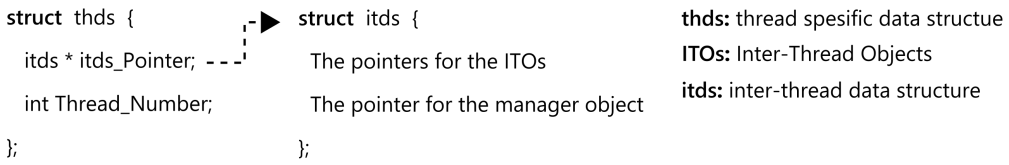


Figure 6: Data structures used in object sharing between the threads

In the figure, the data structure which is depicted as "*itds*" represents the "inter-thread data structure" and the data structure which is depicted as "*thds*" represents the "thread-specific data structure". From the figure, it can can be clearly seen that inter thread data structure holds the address of the inter-thread objects and thread specific data structures holds the address of the inter-thread data structures. In fact, the server class includes members which are the instances of these data structures. In the Pcynlitx applications, the threads are controlled by means of the integer numbers which are assigned by the user to the threads. Thus, in run-time, the thread manager object matches these numbers with the ID numbers which are given by the operating system to the threads. The second member of the "*thds*" is used in order to pass these integer numbers to the threads and the pointer member of "*thds*" passes the address of the "*itds*" to the thread's stack.

6 THE STRUCTURE OF THE SERVER CLASS

In the library construction, the server class is written based on the information coming from the user of the Pcynlitx. Therefore, server class includes every tools and object that are needed on the pcynlitx application development. A simplified version of the server class which is written in the library construction process has been shown in below figure. The Activate member function that is shown on the figure is used in thread creations. Thus, parameter passing is performed by means of Activate member function.

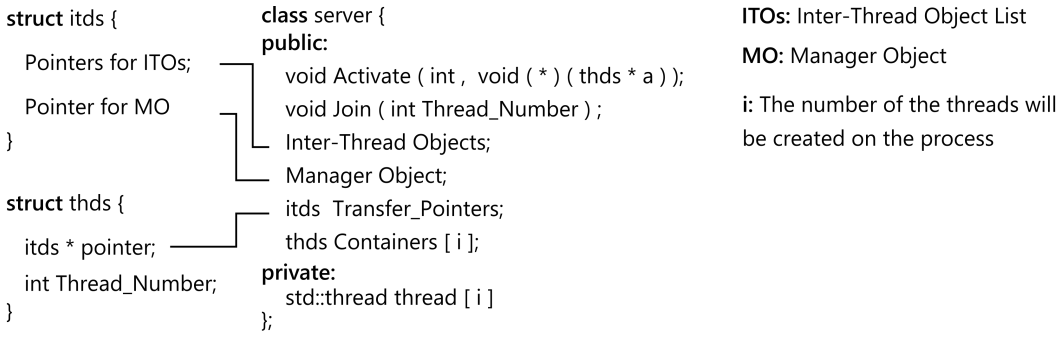


Figure 7: The structure of the server class

6.1 The parameter passing in pcynlitx

Thanks to the data members which are defined on the server class in the types "*itds*" and "*thds*", passing the addresses of the inter-thread objects to the newly created threads is simple. You can easily realise that Active member function takes thread-specific data structure as its argument. Therefore, at first, inside the constructor of the server class, the addresses of the inter-thread objects are assigned to the corresponding pointer members of the "itds". Then, on the creation of each thread, the address of the inter-thread data structure is assigned to the pointer members of the thread-specific data structures which are defined in an array form on the server class. Finally, the addresses of these array elements are passed to the newly created threads (*each element of the array is passed to the different threads*). These operations have been depicted on the below figure.

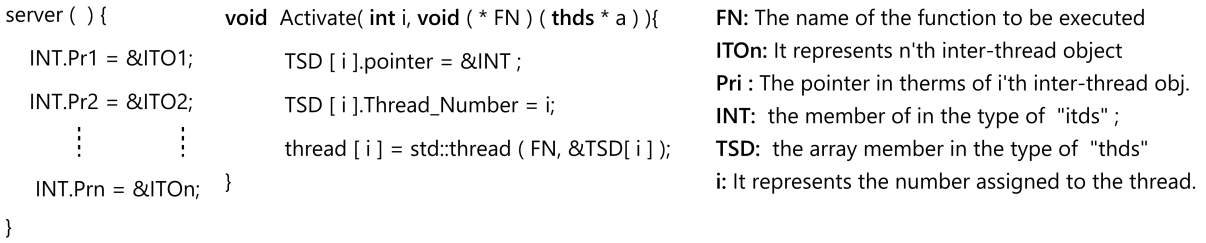


Figure 8: How the addresses of the IT objects are passed

In this project, C++ std::thread library is used for the thread creation and the thread creation is performed by a member function of the server object. In Pcynlitx, this member function is named as "Activate( )". In fact, many functions that are written by Pcynlitx can be considered as wrappers of the standard C++ libraries and/or variables.

7 HOW THE INTER-THREAD OBJECTS ARE USED IN PCYNLITX

In the Pcynlitx, instead of raw pointers, a different indirection mechanism is used for the operations to be performed between the threads. For each one of the inter-thread classes, Pcynlitx produces some special classes and the instances of these classes behave like indirection operators and *their member functions are used as indirect function callers.* In the previous sections, these classes has been defined as "*client classes*" and their instances has been named as the client objects. With the help of the "*client objects*", the programmers can perform indirect function calls to the public member functions of the inter-thread objects without using any pointer. Hence, the client classes are build in a special form. An illustration for the client class declarations has been given in below figure.

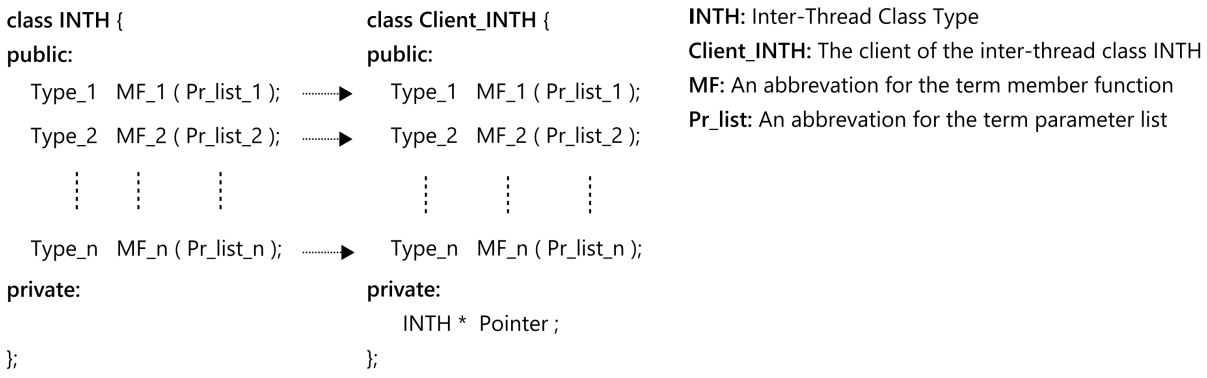


Figure 9: Client class decleration

The declaration of a client class includes an exact copy of each public member function of its counterpart (*the corresponding inter-thread class*) and a private pointer member that points its corresponding inter-thread object. Inside the member functions of a client class, an indirection is performed directly to an instance of its corresponding inter-thread class by means of its pointer member. However, to be able to make this indirection depends on a condition: the address of the inter-thread object must be reachable by the client object. It has been already mentioned that the addresses of the inter-thread objects are passed to the threads in several steps in pcynlitx applications. In fact, the address of the thread-specific data structure (*thds*) is the only parameter of the thread functions executed by the threads and as indicated before, "*thds*" includes a pointer member that points the address of the inter-thread data structure (*itds*) which holds the addresses of the inter-thread objects. Therefore, if the address of the "*thds*" is passed to the thread functions, the addresses of the inter-thread objects will be reachable inside the thread functions scope. This mechanism has been shown in below figure. In here, the main idea is to pass the addresses of the inter-thread objects to the threads based on the information coming from the descriptor file. Strictly speaking, thanks to the description processing system, the Pcynlitx knows what kind of data types will be passed to the threads before the library construction and the client classes are constructed based on this information. Therefore, the constructors of the client classes take the addresses of the thread specific data structures as their arguments and the client objects receive the addresses of their corresponding inter-thread objects from "*thds*" in the object construction. Then, the member functions of the inter-thread objects can be called inside its client. Actually, it can be easily realized that the member functions of the client objects work as indirect function call mechanisms and when they are called, in fact, the member functions of the corresponding inter-thread objects are called indirectly inside the member functions of the client objects.

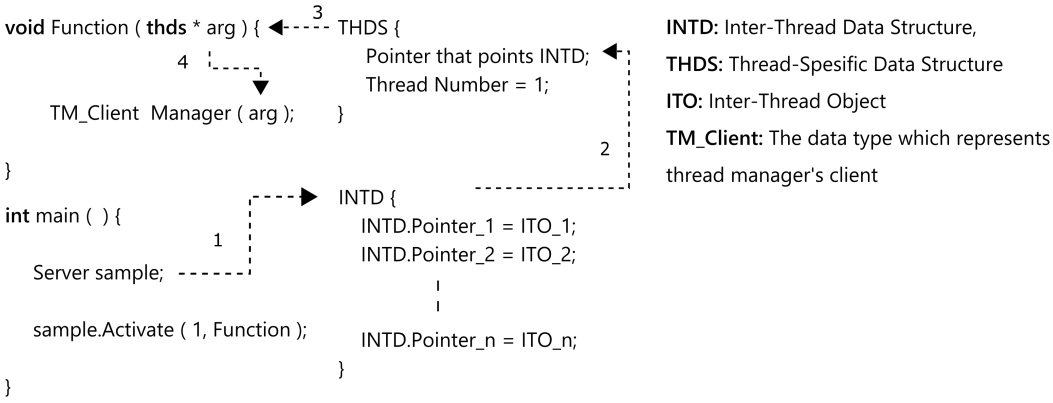
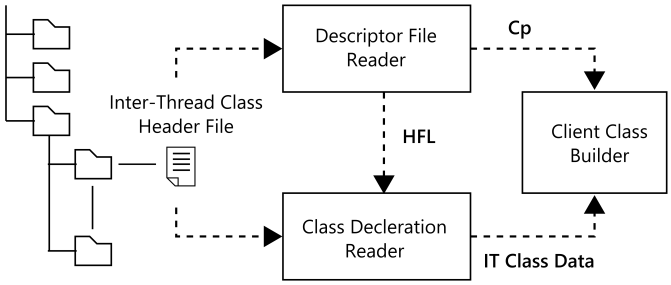


Figure 10: The client class construction

In other words, each member function of an inter-thread object is called indirectly by means of its counterpart that is defined on its client. With the client objects, the programmer can call the member functions of the inter-thread objects by means of usual member function calls.

8 HOW THE CLIENT CLASSES ARE CONSTRUCTED

The meta-program which constructs the client classes (*the Client Class Builder*) has to perform many complex operations in order to produce a client class. To be able to build the client of an inter-thread class, at first, its header file must be read. Therefore, first of all, the locations of the header files including the declarations of the “inter-thread classes” are received from the "Descriptor File Reader".



Cp: The location in which the client class will be constructed
IT Class Data: The collected data about inter-thread class
HFL: The location of the header file of the inter-thread class

Figure 11: Client class builder operations

Then, another sub-meta program "*Class Declaration Reader*" checks the possible syntax errors on the class declaration. After that, it reads the name of the inter-thread class and the trace of each public member function of the inter-thread class from the header file. Finally, the *Client Class Builder* receives inter-thread class data from *Class Declaration Reader* and writes the client class. These steps are illustrated on the figure that is given in above.