Orthogonal Software Architecture Design for Radar Data Processing System with Object-oriented Component and COM Interface

ZHONGZHI LI, XUEGANG WANG, XUELIAN YU
School of Electronic Engineering
University of Electronic Science and Technology of China
Road JianShe, Chengdu, 610054
PR. CHINA
lizz_uestc@163.com

Abstract: - Large scale software system is usually developed by software engineering method, and it needs good architecture and reusable components. Radar data processing system is a complex software system; it needs to complete many tasks such as multi-sensor data fusion, target tracking, data storing and displaying, remote controlling, etc. Based on orthogonal software architecture and component-based software engineering, we propose a new method, orthogonal software architecture with object-oriented component and COM interface in this paper, and we use the proposed method to complete the architecture and components design for radar data processing software system. By eliminating correlation between components, we can improve the reusability and maintainability of component. At the same time, we use COM interface to implement mixed language programming and system integration. After the system development and test, it proves that the new software architecture is reasonable and applicable.

Key-Words: - Orthogonal software architecture; Component-based software engineering; Object-oriented component; Component object model (COM); Module; Radar data processing system.

1 Introduction

With enlargement of software scale and software complexity, the software crisis intensifies day by day. People realized gradually that the system architecture design and the specification explanation become an effective method to improve software productivity and reduce the development complexity and software maintainability, and they are more important than computation algorithm and data structure. The system architecture design plays a pivotal role on the final system's success.

Software engineering is a profession dedicated to designing, implementing, and modifying software so that it is of higher quality, more affordable, maintainable, and faster to build. The term software engineering first appeared in the 1968 NATO Software Engineering Conference, and was meant to provoke thought regarding the perceived "software crisis" at the time [1]. Since the field is still relatively young compared to its sister fields of engineering, there is still much debate around what software engineering actually is, and if it conforms to the classical definition of engineering. Some people argue that development of computer software is more art than science [2], and that attempting to impose engineering disciplines over a type of art is an exercise in futility because what represents good practice in the creation of software is not even defined [3]. Others, such as Steve McConnell, argue that engineering's blend of art and science to achieve practical ends provides a useful model for software development [4]. The IEEE Computer Society's Software Engineering Body of Knowledge defines "software engineering" as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software [5].

In this paper, we apply software engineering thought to design a large-scale software system, radar data processing system. It is a complex system and has many tasks such as multi-sensor data fusion, target tracking, data storing and displaying, remote controlling, etc. And it is inevitable to face mixed language programming, requirement variety, large maintenance, etc. So, we need to consider adequately the component reusability and maintainability; it is important to choose good software architecture and reusable components. We proposed a new method, orthogonal software architecture with object-oriented component and COM interface in this paper for radar data processing system. Firstly, we introduce the software developing theory, especially orthogonal

software architecture and component-based software engineering, then the new software architecture with object-oriented component and COM interface is proposed, and then we describe the requirement of radar data processing system, finally, the software system architecture design using the proposed method is given.

2 Software developing theory

In this paper, we use software architecture (especially orthogonal software architecture) and component-based software engineering (with COM interface) technology to develop software system. So, we describe the related theory firstly.

2.1 Software architecture

Software architecture is originated from the software organization and structure research by Dijkstra [6] and Pamas[7] in the 1960s and 1970s; it is a natural evolution for software design abstract [8]. More and more people recognize the importance of software architecture, and notice that in-depth studying the software architecture is a promising way to improve software productivity and maintenance. There are three periods during software architecture improvement: (1) Mainframe structure, which is used to be suitable with central system. In this structure, customer's data and procedures are concentrated in the host with a few GUI interfaces; meanwhile it is difficult to support remote database access. With the widespread use of computer, the structure has been eliminated. (2) In the middle of 1980s, Client / Server distributed computing structure comes out. The applications are shared in the client (PC machine) and server; requests are usually dealt with relational database; client PC is used to display and implement the business logic after it received processed data. Furthermore, this structure supports modular design and usually has GUI interfaces, and its flexibility makes the structure to be widely used. However, in large-scale software system, the structure is not good enough for system deployment and extension. (3) Thereafter, three-layer / multi-layer architecture appears which is based on Internet and web applications. In the three-layer architecture, the client (request for information), procedures (deal with requests) and data (to be operated) are physically isolated. Three-layer structure is a more flexible architecture, it extracts the display logic out of the business logic, that means business logic is independent in coding, you do not need to care about how and where to show them. Business logic layer is in the middle, so it does not need to concern where the data come from and how to display, and is independent with background system. Considering these, three-layer structure has a better portability, supports different working platform, and allows user requests load balanced between several servers. Also it has a strong security, because the application has been isolated with the client customer. Application server is a component of the three-layer / multi-layer architecture and it is located in the middle layer.

Software architecture is still in development, its academic definition has not yet unified. Here are some typical definitions: Garlan &. Shaw model [9] and CFRP model [10] emphasize that the architecture is composed of components, connectors and its constraint (or connection semantic), that is, from the composition view to look at the software architecture. Perry &. Wolf model [11] and Vestal model [12] focus on architecture style, model and rules etc., it would like to use an overlook view to consider software architecture. Definitions from IEEE610.12 - 1990 [13] emphasize not only the basic composition, but also the environment of architecture (the interaction with the outside world). Boehm model [14] emphasizes that the software architecture is a set of concepts and decisions on software system design; it will help a developing system satisfy an important function and quality requirement. Bass, Ctements &. Kazman model [15] [16] ignores the software architecture details, and pays attention to the abstract conception of software system and the overall characteristics (the external visual attributes). Although from different view points, all the software architecture models considered the systems structure, and mentioned the following entities: components, interaction relationship between the components, constraint, and topology consisted by connectors and components, design principles and guidelines.

2.2 Orthogonal software architecture

Orthogonal software architecture is composed of the organization layers and clues. Layers are constituted by a group of the same abstract level components. Clue is a particular case in the sub-system, it is constituted by components which are used to complete functions of the different levels (by calling each other to correlate); each clue completes part of the system relatively independent functions. There is no or few correlation between clues' implementation. In the same layer, components are not allowed to call each other [17]. If the clues are independent, that is, the different components in different clues do not call each other, and then the structure is orthogonal. From the above

definition, we can see that the orthogonal software architecture is a hierarchical structure based on the vertical components of the clues. Its basic idea is dividing the application structure into a number of vertical clues (sub-systems) according to the function orthogonal correlation. Clues are divided into several levels, and each clue is composed of several components with different abstract levels and functions. The components with the same level have the same level of abstraction in every clue. Therefore, we can get orthogonal software architecture main features: (1) Constituted by *n* (*n*>1) clues (sub-systems); (2) System has *m* (*m*>1) different levels abstraction layer; (3) Clues are independent (orthogonal); (4) System has a common drive layer (usually the highest layer) and common data structures (usually the lowest layer).

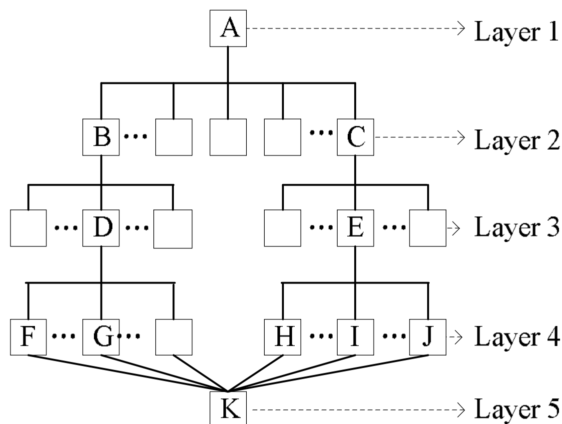

Fig.1. Orthogonal software architecture framwork

In a large-scale and complex software system, its first level sub-clues can be divided into lower level sub-clues (the second level sub-clues) and formed the multi-levels orthogonal structure. Fig.1 shows an orthogonal software architecture framework which is composed of three level clues and five layer structure. ABDFK is a clue and ACEJK is another clue. Here, B and C are in the same level, so they will not allow to be called by each other. In general, the fifth layer is a physical database connector or equipment component, and it will be used by the whole system.

In the evolution process of software, system requirement are always changing. Because of the orthogonal, every requirement variety only affects one clue. By this way, orthogonal software architecture localizes the variety and influence. Here are the advantages: (1) Clear structure, easy to understand. The expression of the orthogonal software architecture is easier for user to understand. For clues are independent and can not call each other in the same layer, the structure is simple and clear. The position of a component has already told us it's abstract and functions. (2) Easy to modify and maintain. For each clue is independent, one clue's variey will not affect others. When system requirement variety occurs, we can divide them into sub requirement, then deal with them from the clue and components view. That means if we need to add or delete functions in a system, we only need to add or delete the correlated clues, and the whole system structure will not be affected. (3) Easy to reuse, and has strong portability. Because orthogonal software architecture can be shared by all the applications if they have the same or similar layers and clues in a domain.

But in orthogonal software architecture, components usually have strong correlation in the same clue, that is, components are hard to be changed and maintained when the clue changes. In order to solve this problem, we propose object-oriented method in this paper to eliminate or weaken the correlation between components in the same clue.

2.3 Component-based software engineering

An individual component is a software package or a module that encapsulates a set of related functions (or data). All system processes are placed into separate components so that all of the data and functions inside each component are semantically related (just as with the contents of classes). Because of this principle, it is often said that components are modular and cohesive. With regard to system-wide co-ordination, components communicate with each other via interfaces. When a component offers services to the rest of the system, it adopts a provided interface which specifies the services that can be utilized by other components and how. This interface can be seen as a signature of the component - the client does not need to know about the inner workings of the component (implementation) in order to make use of it. This principle results in components referred to as encapsulated. The UML illustrations within this article represent provided interfaces by a lollipop-symbol attached to the outer edge of the component.

However when a component needs to use another component in order to function, it adopts a used interface which specifies the services that it needs. Another important attribute of components is that they are substitutable, so that a component can replace another (at design time or run-time), if the successor component meets the requirements of the initial component (expressed via the interfaces). Consequently, components can be replaced with either an updated version or an alternative for

example, without breaking the system in which the component operates. As a general rule of thumb for engineers substituting components, component B can immediately replace component A, if component B provides at least what component A provided, and uses no more than what component A used. Software components often take the form of objects or collections of objects (from object-oriented programming), in some binary or textual form, adhering to some interface description language (IDL) so that the component may exist autonomously from other components in a computer.

When a component is to be accessed or shared across execution contexts or network links, techniques such as serialization or marshalling are often employed to deliver the component to its destination. Reusability is an important characteristic of a high-quality software component. A software component should be designed and implemented so that it can be reused in many different programs. It takes significant effort and awareness to write a software component that is effectively reusable. The component needs to be:

- fully documented
- thoroughly tested
  - robust - with comprehensive input-validity checking
  - able to pass back appropriate error messages or return codes
- designed with an awareness that it will be put to unforeseen uses

In the 1960s, programmers built scientific subroutine libraries that were reusable in a broad array of engineering and scientific applications. Though these subroutine libraries reused well-defined algorithms in an effective manner, they had a limited domain of application. Commercial sites routinely created application programs from reusable modules written in Assembler, COBOL, PL/1 and other second- and third-generation languages using both System and user application libraries. Modern reusable components encapsulate both data structures and the algorithms that are applied to the data structures. It builds on prior theories of software objects, software architectures, software frameworks and software design patterns, and the extensive theory of object-oriented programming and the object oriented design of all these. It claims that software components, like the idea of hardware components, used for example in telecommunications, can ultimately be made interchangeable and reliable. On the other hand, it is argued that it is a mistake to focus on independent components rather than the framework without which they would not exist.

2.3 COM Technology

Component Object Model (COM) [18] is a binary-interface standard for software componentry introduced by Microsoft in 1993. It is used to enable interprocess communication and dynamic object creation in a large range of programming languages. The term COM is often used in the Microsoft software development industry as an umbrella term that encompasses the OLE, OLE Automation, ActiveX, COM+ and DCOM technologies. The essence of COM is a language-neutral way of implementing objects that can be used in environments different from the one in which they were created, even across machine boundaries. For well-authored components, COM allows reuse of objects with no knowledge of their internal implementation, as it forces component implementers to provide well-defined interfaces that are separate from the implementation. The different allocation semantics of languages are accommodated by making objects responsible for their own creation and destruction through reference-counting. Casting between different interfaces of an object is achieved through the QueryInterface() function. The preferred method of inheritance within COM is the creation of sub-objects to which method calls are delegated.

Although the interface standard has been implemented on several platforms, COM is primarily used with Microsoft Windows. For some applications, COM has been replaced at least to some extent by the Microsoft .NET framework, and support for Web Services through the Windows Communication Foundation (WCF). However, COM objects can be used with all .NET languages through .NET COM Interop. Networked DCOM uses binary proprietary formats, while WCF encourages the use of XML-based SOAP messaging. COM is very similar to other component software interface technologies, such as CORBA and Java Beans, although each has its own strengths and weaknesses. The characteristics of COM make it most suitable for the development and deployment of desktop applications, for which it was originally designed.

COM programmers build their software using COM-aware components. Different component types are identified by class IDs (CLSIDs), which are Globally Unique Identifiers (GUIDs). Each COM component exposes its functionality through one or more interfaces. The different interfaces supported by a component are distinguished from each other using interface IDs (IIDs), which are GUIDs too. COM interfaces have bindings in several languages, such as C, C++, Visual Basic, Delphi, and several of

the scripting languages implemented on the Windows platform. All access to components is done through the methods of the interfaces. This allows techniques such as inter-process, or even inter-computer programming (the latter using the support of DCOM). Component Object Model (COM) specifies architecture, a binary standard, and a supporting infrastructure for building, using, and evolving component-based applications. It extends the benefits of object-oriented programming such as encapsulation, polymorphism, and software reuse to a dynamic and cross-process setting. Distributed COM (DCOM) is the distributed extension of COM. It specifies the additional infrastructure that is required to further extend the benefits to networked environments.

3 Orthogonal software architecture with object-oriented component and COM interface

Based on orthogonal software architecture and component-based software engineering, we propose a new method: orthogonal software architecture with object-oriented component and COM interface.

3.1 Object-oriented component

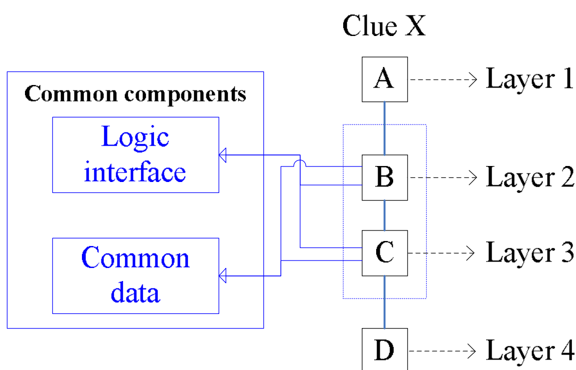

Fig.2. Object-oriented component

Correlation between components is the biggest problem in components reusability. We define correlation as data correlation or logic correlation. When software requirement occurs (data change or business logic change), we have to update all correlated components; that is an annoying thing for developers. Therefore, we need to eliminate or reduce correlation between components as much as possible, i.e. making component to be completely independent (not correlated with other components in the same clue). We set the principles as follows:

- Components in the same layer must be uncorrelated.
- Components (in the same clue) in the different layers can not correlated directly, if happened, we generate the new base component by object-oriented method.

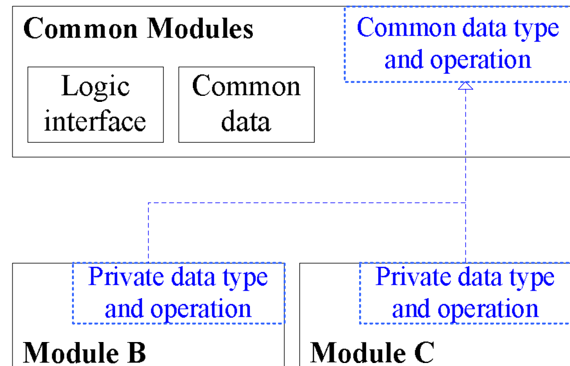

Fig.3. Base class and derivation

In Fig.2, we assume that components B, C in clue X are correlated. When software requirement variety occurs (data change or business logic change), we have to update components B, C simultaneously. In order to make the components to be independent, we apply object-oriented ideology to process them. Usually, correlation exists in data and business logic; firstly we generate new common components, common data component and common business logic interface component from components B, C, then components B, C inherit from the common components shown in Fig.3. Components B, C only implement their private data type and operation, so they will be independent. The added common components maybe are a little complex, but the developing cost of new common components is always less than original components correlation.

3.2 Component with COM interface

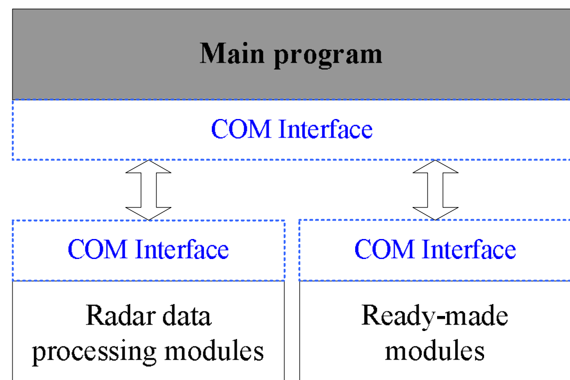

Fig.4. Object-oriented component

The large-scale software system design usually

uses the software engineering modular thought. This kind of developing method needs the cooperation of many developers who usually use different development tools and different programming languages, and sometimes we also need to integrate ready-made modules developed by different tools. So, it needs mixed language programming, and the COM technology is an effective way to solve the integration requirement.

In Fig.4, main program (executive program), radar data processing modules and ready-made modules are developed with different programming tools, and they can not communicate each other. So, we need add COM interface to integrate them. As an example, in radar data processing software system, we develop the radar data processing modules with Visual Studio .Net and develop the main program with Visual C++6.0, and we give the implementation of COM interfaces as follows.

(1) COM server creation in .Net

- COM interface definition and implementation

In order to allowing COM to access the properties, methods and events, we have to define them with DispId property in the class interface and implement them in the class. The sequence of these members defined is their sequence in COM. To open COM, interfaces must be public in the class. There need a GUID feature before class name and interface name. An Example is given as follows:

```
namespace RadarDataProcessLib
{
   public class ComInterface
   {
      // COM methods definition
      [ComVisible(true), Guid("6F67D6BD-B902-4fc8-8C19-9B2FD0A61802")]
      public interface IComRadarDataProcess
      {
         // method 1 definition
         [DispId(1)]
         void method_1();
         ……
      }
      // COM events definition
      [ComVisible(true), Guid("CA7C94D7-79F6-472f-A9DF-4F7D71881519")]
      [InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
      public interface IComRadarDataProcessEvent
      {
         // event 1 definition
         [DispId(1)]
         void OnEvent_1();
         ……
      }
      // COM class definition
      [ComVisible(true), Guid("BD9771CC-78D4-49bb-9EFE-2321411BF97F"), ClassInterface(ClassInterfaceType.None)]
      [ComSourceInterfacesAttribute(typeof(IComRadarDataProcessEvent))]
      public class ComRadarDataProcessDll : IComRadarDataProcess
      {
         // Methods and events implement defined in interface
         ……
      }
   }
}
```

- Interaction between COM objects and management application

Before creating COM objects, we must register them for COM Interop. It set the value of “Register for COM Interop” as true. For COM objects that can be called by external objects, libraries must have a strong portfolio name. It needs to use SN.EXE to generate a name key, run the command as follow:

```
sn -k ComServerDll COM Key.snk
```

(2) COM interface reference in VC++6.0

- Add ATL support to project
- Import component library to project

```
#import "COMIntfLib.tlb" named_guids, no_namespace
```

- implement events receptive object

Use IDispEventImpl template in ATL to implement event receptive object, and use SINK_ENTRY_EX micro add event processing articles in event sink mapping table as follows.

```
class EventReceiver :  public IDispEventImpl<IDC_EventReceiver,EventReceiver,
      &__uuidof(IComRadarDataProcessEvent),&LIBID_COMIntfLib, 1, 0>
{
public:
      EventReceiver()
      {
            // Do nothing
      }

      // Event sink mapping table declaration,
      BEGIN_SINK_MAP(EventReceiver)
            SINK_ENTRY_EX(IDC_EventReceiver, __uuidof(IComRadarDataProcessEvent), 1, OnSrcPointReceived)
            ……
      END_SINK_MAP()

      // incoming calls from the C# Program
      STDMETHOD_(void,OnSrcPointReceived)(BSTR parameter);
      ……
protected:
      IComRadarDataProcess* m_pICM;
};
```

4 Software architecture design for radar data processing system

In this paper, we design the radar data processing system's architecture based on orthogonal software architecture with object-oriented component and COM interface. We divide components by clues (function classification) and implement them with object-oriented method to keep independence and reduce correlation in the same clue. At the same time, we add COM interface to modules for mixed language programming and system integration. The proposed method is beneficial to system expansion and maintenance, and it improves the system reusability, scalability, and maintainability.

4.1 System requirement

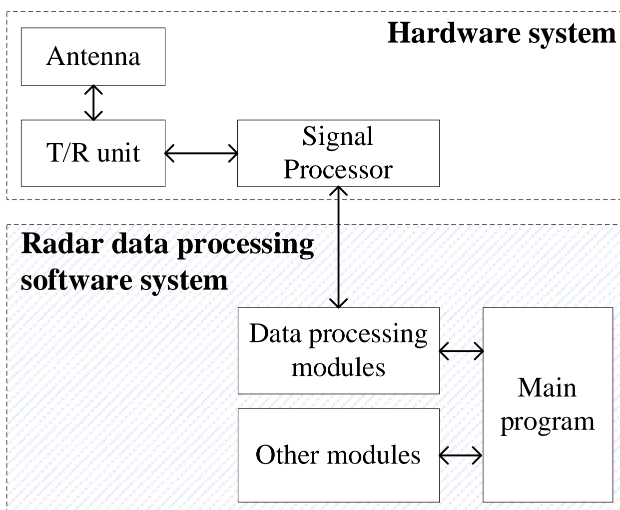

Fig.5. Systerm topology

Radar data processing software system is a part of the radar surveillance system and their relationship is shown in Fig.5. Data processing modules implement tasks associated with the radar data such as data communication, target tracking, target identification, primary and secondary radar data displaying, data management and replaying, radar control, and so on. The main program completes the system integration. Because the system function is complex, we need to design the system with software engineering modular thought. The radar data processing software system discussed in this paper uses two development tools: Visual Studio .Net and Visual C++6.0. Data processing modules are developed as dynamic link library (DLL) by Visual Studio .Net, and the main program is developed by Visual C++6.0. Except the data processing modules, there have some other modules such as electronic maps, statistical analysis, and so on. (These modules are ready-made or developed by different tools). Because of the difference between modules, they cannot interact directly, and the COM technology is the method for mixed language programming.

The system deployment is shown in Fig.6. It works in a distributed network environment, and includes local controller, remote controller, data server and surveillance terminal. Their functions are as follows.

(1) Local controller:

- Communication with signal processor.
- Receive primary and secondary radar data.
- User-defined protocol parsing.
- Primary and secondary radar data displaying.
- Communication with remote controller.

- Radar controlling.

(2) Remote controller:

- Communication with local controller.
- Communication with other interfaces.
- Multi-sensors data fusion.
- Target tracking.
- Track storing and management.
- Primary and secondary radar data displaying.
- Data storing and accessing.
- Target tracking replaying.
- Radar remote controlling.

(3) Surveillance terminal software functions:

- Communication with controller.
- Database accessing ability.
- Track displaying and replaying.

(4) Data server:

- Data storing and accessing interface.

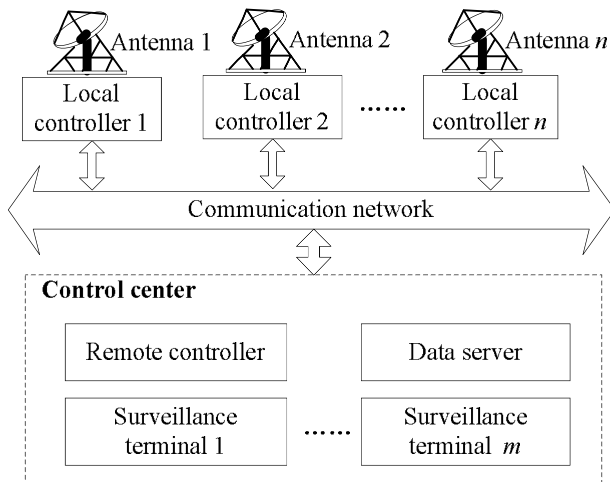

Fig.6. Systerm deployment

4.2 System layers design

Software requirement is usually dynamic. In order to cope with the influence caused by the possible variety in the structure and to improve reusability and maintainability, we use orthogonal architecture for software design. Firstly, we design the layer structure of the system as shown in Fig.7, and the system is divided into five layers: interface layer provides COM interface for mixed language programming; application layer provides GUI and man-machine interaction interfaces; business logic layer provides kernel algorithms such as original data pre-processing, clutter processing, track processing (target points aggregating, track initializing, maintaining, and discarding), target recognition according to the number, range, speed of targets; data layer provides interfaces for accessing the database, text and the memory data structure; driver layer provides interfaces to access hardware such as CPCI cards. Then we divide system to many components after determination of clues such as track processing, data storing, signal processor controlling, etc.

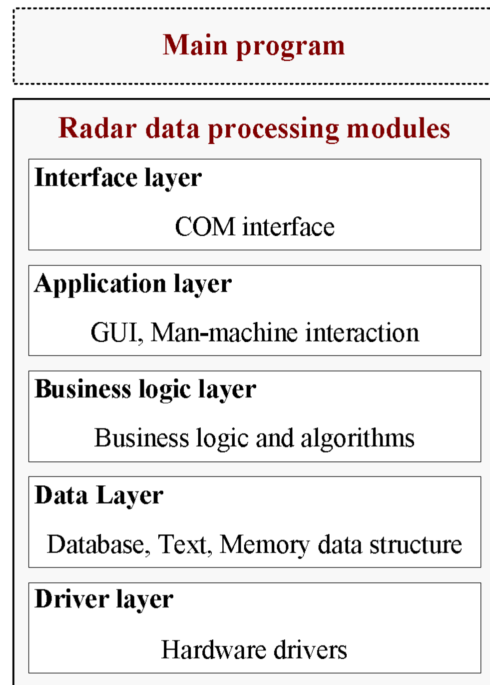

Fig.7. Systerm layer structure

4.3 System components design

After determination of system layers and clues, we design the radar data processing system software architecture as shown in Fig.8.

As an example, in track processing clue linked by blue line, we explain how to generate object-oriented component. CPCI driver module obtains the radar data from signal processor through CPCI card; the data is stored in Memory data structure module; Primary radar data fusion module obtains the data and complete radar data pre-processing, then Track processing module complete target tracking, target identification, etc. At last, Track display module provides GUI interface for operator. Because all these modules are correlated with memory data and business logic task dispatching, it makes the system difficult to maintain when software requirement variety occurs. In order to fix this problem, we use object-oriented method to create a common data component and common logic component (they compose the common modules in red color: System task dispatching and Common data modules), and then modules in track processing clue inherit from them. These object-oriented components do not need to care about different data structure and business logic, so they are uncorrelated, and easy to reuse.

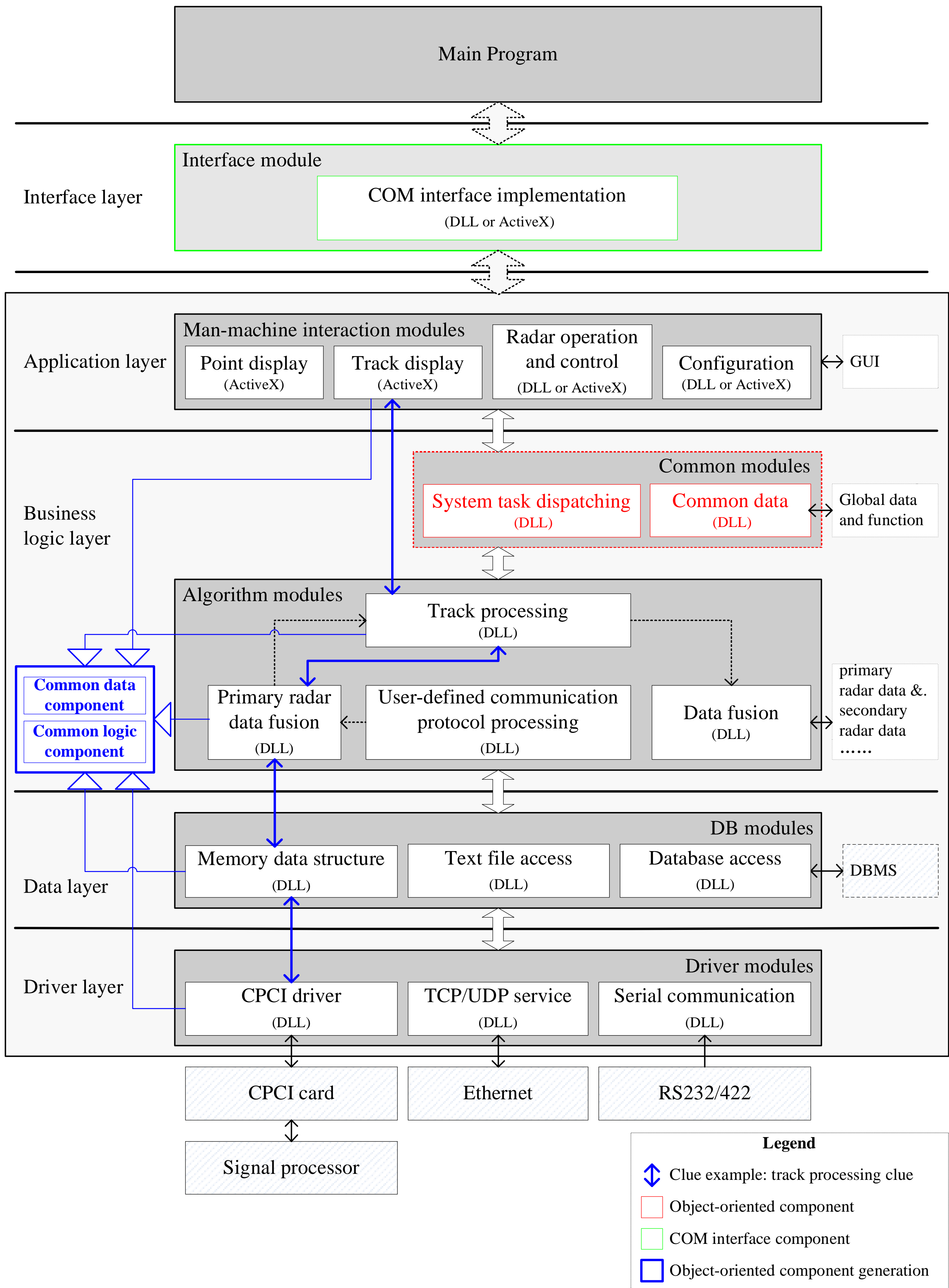

Fig.8. Systerm componets architecture &. object-oriented componet design example

5 Conclusion

In this paper, we analyze the orthogonal software architecture and the component-based software engineering. Then we propose the idea of object-oriented component with COM interface to eliminate the correlation between the components and to integrate mixed language programming modules. According to the proposed method, we design the radar data processing software system. In the process of development, software requirement variety occurs regularly, and we confine the variety into common modules by object-oriented component. Therefore, the system is easy to maintain and expand, and the architecture is applicable for large-scale software development. There is a minor disadvantage that the system task dispatching module is complex because it is correlated with various clues (with various components), but it is acceptable when most components become independent.

Acknowledgment:

This work was supported in part by the National Natural Science Foundation of China (No. 60736045).

References:

[1] P. Naur and B. Randell, *Software engineering: Report of a conference sponsored by the NATO Science Committee*, Garmisch, Germany: Scientific Affairs Division, NATO, 1968.

[2] Hey, Programmers, We Got No Theory! *Dr. Dobbs Journal*, March 22, 2010 (Retrieved March 26, 2010)

[3] Ivar Jacobson and Ian Spence, Why We Need a Theory for Software Engineering, *Dr. Dobbs Journal*, October 02, 2009.

[4] McConnell Steve, *The Art, Science, and Engineering of Software Development*, IEEE Software, Vol.15, No.1, 1998.

[5] Alain Abran, James W. Moore, Pierre Bourque, and Robert Dupuis, *Guide to the Software Engineering Body of Knowledge - 2004 Version*, IEEE Computer Society, 2004, pp. 1–1.

[6] E.W. Dijistra, The structure of the multi-programming operating system, *Communication of ACM*, Vol.11, No.5, 1968, pp. 341-346.

[7] D.L. Pamas, P.C. Clements, and D.M. Weiss, *The modular structure of complex systems*, In: Proeeedings of the 7th intemational Conference on Software Engineering (Orlando, Florida, USA, March 26-29, 1984), IEEE Press, Piscataway, NJ, 1984, pp. 408-417.

[8] Sun changai, Jin maozhong and Liu chao, Overviews on Software Architecture Research, *Journal of software*, Vol13, No.7, 2002, pp. 1228-1237.

[9] D. Garlan and M. Shaw, *An introduction to software architecture*, Technique Report, CMU/SEI -94-TR-21 ， Camegie Mellon University, 1994.

[10] The Boeing ComPany-Defense and Space Group, *STARS conceptual ftamework for reuse Processes，lockheed martin tactical defense system*, STARS Program Technieal Report, 1994.

[11] D.E. Perry, *Software engineering and software architecture*, In: proceedings of the Intemational Confereneeon Software: Theory and Practiee, Beijing: Eleetronic Industry Press, 2000, pp. 1-4.

[12] S. Vestal, *A cursory overview and comparison of four architecture description languages*, Honeywell Technology Center Technical Report, 1993.

[13] IEEE, *IEEE Glossary of Software Engineering Terminology*, 610.12-1990, 1998.

[14] C. Gacek, A. Abd-Allah and B.K. Clark, et al., *On the definition of software system architecture*, In: Proceedings of the lst Intemational Workshop on Architecture for Software Systems, New York: ACM Press, 1995, pp. 85-95.

[15] Camegie Mellon University Software Engineering Institute, *How do you define Software Architecture*, http://www.sei.cmu.edu/ architecture/definitions.html.

[16] Rick Kazman, *Software Architecture*. in handbook of software Engineering and Knowledge Engineering, S-K Chang(ed.), World Scientific Publishing, 2001.

[17] Y.S. Zhang, *Software Architecture*, Beijing: Tsinghua University Press, 2004.

[18] Microsoft Corporation, *The Component Object Model Specification*, http://www.microsoft.com/ com/default.mspx, 2010.