4.2.3.1 Modularity

Return to Maintainability

- ☐ **[char]Please Review**

About

Return to Top

Modularity describes a characteristic of a system or program to be organized into smaller, reusable components. Each component is self-contained providing an interface that describes the functionality it offers to other components of the system. It optionally provides of a set of interfaces it requires in order to fulfill its functionality. In Object Orient programming, the functionality is encapsulated as set of data attributes. The Module exposes access to the data attributes by defining public interfaces that other components can call to manage and manipulate the data attributes of the object (i.e., Module). If the component relies on other components, then it can specify the required components (i.e., Modules).

Many Modules are described by files that only describe the interface to the Module and contains no actual functionality. For example, in C/C++, these Module interfaces are described using header files (i.e., `.h`, `.hpp`) files; in Java the file is a regular `.java` file but contains the key word `interface` in the class descriptor; in Common Object Request Broker Architecture (CORBA), interfaces are described as stubs. ECMAScript (i.e., JavaScript), PHP, etc. uses the concept of Duck Typing at runtime to accomplish the equivalent of interfaces (i.e., it is based on the methods defined within a Module at runtime rather than on a static, abstract fixed interface).

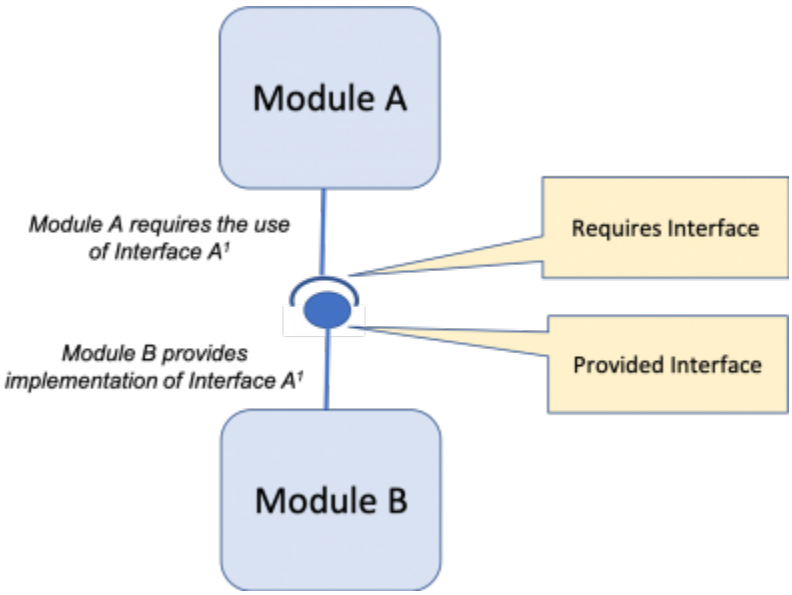


Figure 1: Modeling Modules Provided and Required Interfaces.

Regardless of kind of implementation (i.e., interface, stubs, DuckTyping, etc.), APIs need to be public, formalized and freely available to the general public at no cost. The benefit of using interfaces, stubs, or

headers is that un-implemented or mismatched functionality can be checked at compile or link time. Dynamic Plug In interfaces (i.e, DuckTyping) can only be checked at runtime. However, dynamic type is very definitely modular. One way around the run-time error is to always provide a default implementation.

The use of Modules allows for the architecture and design of a system or program to be re-factored by extracting functionality into new Modules, combining Modules into a new Module, or composing Modules from other Modules. Often the rules for refactoring a system of a program follow many of the same rules as data Normalization.

Yiming et al.[1)] introduce complex network theory into software engineering to develop a metric for measuring Modularity. They analyze existing software by representing it as a network using feature coupling network (FCN). Each method and attribute are considered a node in the node network. Couplings between methods and attributes are considered edges. Edge Weight represents a weighted value based on the number of invocation paths for the node (i.e., method 'A' is called 2 times, is a weight of '2', attribute is used once, gets a weight of '1').

> FCN = (N, E, Ψ)
> **Where:**
>
> - FCN : Feature Coupling Network
> - N : the Node set (number of attributes or methods)
> - E : the Edge set (couplings between Edge Nodes) of
> - Ψ : the Edge Weight (number of invocation paths)

Yiming et al., apply the Weyuker's criteria which is widely used in the field of software metrics, to validate the modularity as a software metric theoretically, and also perform an empirical evaluation using open-source Java software systems to show its effectiveness as a software metric to measure software modularity.

DDS Specifics

Return to Top

Although the Modularity was primarily developed to look at software, it can also be applied to distributed systems by changing the definition of 'N' from the number of attribute or methods to the number of Publishers and Subscribers , and the 'E' to the number of couplings between the publishers and subscribers. 'Ψ' would be a weighting for the volume of couplings between the publishers and subscribers.

In Practice this means, that centralizing functionality into a normalized set of Endpoints, that couplings are the connections between the endpoints and the volume of traffic represents a weighting, then we are reduced to three variables: N, E and Ψ. This comes down to the age old question: is it better to have one command with 100 options, 100 commands with no options or a set of commands that can share a

restricted set of options. Its obvious that the first two choices are unacceptable. The key comes down to the last option where the options are all related based on the functionality of the command.

data_distribution_service_dds is about Publishers and subscribing data. The publishers and subscribers represent the Node Set, the Edge Set is represented by the Topics and the edge weight represents the volume of traffic. The same question needs to be asked. Is it best to have one messages with a 100 attributes in the data structure, or 100 messages with no data attributes? Obviously the answer is somewhere between where each message is built around a data structure that is formulated on functionality.

1)

Yiming Xiang, Weifeng Pan, Haibo Jiang, Yunfang Zhu, Hao Li, Measuring Software Modularity Based on Software Networks, 14 February 2019, Entropy, Accessed 3 Aug 2020, https://www.mdpi.com/1099-4300/21/4/344/htm

From:
https://omgwiki.org/dido/ - **DIDO Wiki**

Permanent link:
https://omgwiki.org/dido/doku.php?id=dido:public:ra:1.4_req:2_nonfunc:20_maintainability:modularity&rev=1605396917

Last update: **2020/11/14 18:35**