4.3.3.1 Modularity

Return to Maintainability

- ☐ **[char]Please Review**

About

Modularity describes a characteristic of a system or program to be organized into smaller, reusable components. Each component is self-contained and provides an interface that describes the functionality it offers to other components of the system. It optionally provides a set of interfaces required to fulfill its functionality. In Object-Oriented Programming (OOP), this functionality is encapsulated as set of data attributes. The Module exposes access to these data attributes by defining public interfaces 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 contain no actual functionality. For example, in C/C++, these Module interfaces are described using header files (i.e., `.h`, `.hpp`) files; in Java these files are regular `.java` files but contain the key word `interface` in their class descriptor; in Common Object Request Broker Architecture (CORBA), interfaces are described as stubs. ECMAScript (i.e., JavaScript), PHP, etc. use 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).

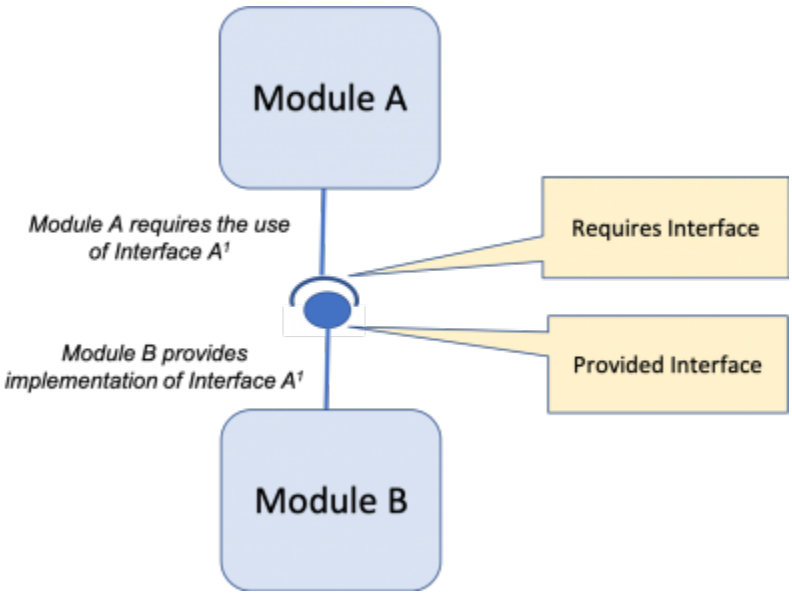


Figure 1: Modeling Modules with Provided and Required Interfaces.

Regardless of the 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 run-time errors is to always provide a default implementation.

The use of Modules allows 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] introduced the use of complex network theory into software engineering with which to develop metrics for measuring Modularity. They analyzed existing software by representing it as a network using a feature coupling network (FCN). Each method and attribute is 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, has a weight of '2', each 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 Weyuker's criteria, which are widely used in the field of software metrics, to validate modularity as a software metric theoretically, and also to perform an empirical evaluation using open-source Java software systems to show its effectiveness as a software metric to measure software modularity.

DIDO Specifics

Return to Top

Although 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 attributes or methods to the number of Publishers and Subscribers, and E to the number of couplings between the publishers and subscribers. Ψ would then be a weighting for the volume of couplings between publishers and subscribers.

In practice this means that if functionality is centralized into a normalized set of Endpoints such that couplings become the connections between endpoints and the volume of traffic represents a weighting, the end result is that we are reduced to three variables: N, E and Ψ. Thus, we must address 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? It's obvious that the first two choices are unacceptable so we are left with with the last choice where the options are all related based on the functionality of the command.

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=1622584245

Last update: **2021/06/01 17:50**