Thoughts about Modular Programming

By J.A.J. van Leunen

Last modified: 20 februari 2016

Abstract

Modular programming is a very efficient way of system creation. By reducing the number of relevant relations the method diminishes the complexity of configuring and supporting modular systems. The method uses the available resources in an optimal way.

The current way of software generation uses an object oriented way of system construction that does not encapsulate the objects, such that their internals are effectively hidden and guarded against obstructive access by external objects. This paper introduces a new way of notation that enforces this encapsulation. Many programming languages already implement part of the required methodology. An example is the razor language. This paper extends these ideas to a modular way of programming.

The approach makes from every relational database a modular database and from every file system a modular file system. It makes from every communication service a modular communication service and it standardizes programming such that reuse can be optimized in a global way. It will improve the robustness and reliability of software and enables to a large extent automated system configuration.

Introduction

One of the most universal laws is not $E=m\,c^2$, but instead it can be phrased as a commandment: "Thou shalt construct in a modular way". The reason is that modular construction is by far the most efficient way of constructing complex systems. Simple constructs can easily be managed, but realizing very complex systems is hampered by the quick increase of the complexity of the relational structure of the construct. Modular construction encapsulates the modules such that relations that are only relevant in the internals of the module are hidden from the outside of the module. To the outside modules have public properties and public methods. Access to the properties is arranged via get and set methods in such a way that the state of the module cannot be corrupted by the external access of this interface. The modules can have other methods that accept commands and/or data. The interface is published in a type definition. Types are arranged in classes and such that the modules can be coupled in a standardized way. Modules can occur in objects, which contain series of equal or different modules. Compared to monolithic construction, can modular construction reduce relational complexity of complex systems with several orders of magnitude! Working with modules requires a special notation.

Notation

A module has a type and a name. We indicate the module and its name with special symbols ⫷ and ⫸. These are characters that you will not find on your keyboard, but you can easily enter these in your text by using the Unicode input utility. That utility can be downloaded from http://www.fileformat.info/tool/unicodeinput/ . The two characters have hexadecimal Unicode identifiers 2af7 and 2af8.

A module with name 'special_1' will be indicated as ⫷ special_1⫸. The type of ⫷ special_1⫸ is defined by the type definition ≪special≫, which defines a category of ≪special≫ modules of which ⫷ special_1⫸ is a member. Properties of members of a category can be get by special get methods, which are defined in the type definition and are used on the individual model as ⩽ special_1⩾.getProperty_1(). Here Property_1 is just an example. This property has a data type, which is defined in the style sheet ⪻special⪼.Property_1 of ⪻special⪼. The get method is described in the style sheet ⪻special⪼.getProperty_1(). Similarly a set method may be available and is described in ⪻special⪼.setProperty_1(p). Here p is an item that has type ⪻special⪼.Property_1 . The interface defines all public methods.

The characters ⪻ and ⪼ have Unicode's 2abb and 2abc.

An object is a series of equal or different modules of which the members can be accessed by combinations of type names and individual names, such as in the example:

【Object_sample】 = 〖⪻special⪼.⩽ special_1⩾, ⪻another⪼.⩽ anyName⩾,

⪻another⪼.⩽ anyOtherName⩾〗

Here we used for the special brackets the Unicode's 3010, 3011, 3016 and 3017.

Modules can be constructed from other modules and from objects that contain series of objects. Elementary objects do not contain other objects. Modular systems are not contained in an encapsulating module, but they can cooperate in a community in which they cooperate.

Storage

Elementary modulus contain data that must be stored. The storage service works module oriented and uses the get and set methods of the modules in order to communicate between the storage medium and the modules. Such storage services are module oriented databases. Data are stored in a compressed and an encrypted way. Each location has an individual storage service. The type definitions must be present at every location where the corresponding modules are active. Apart from defining the type the construct supports creation and annihilation of the corresponding modules. It will be clear that the type definition service is closely linked with the storage service.

Creation and annihilation

Elementary modules can be created and annihilated. The corresponding type system provides these services. Thus

⩽ special_1⩾ = ⪻special⪼.create('special_1', dataStream)

creates a new module ⩽ special_1⩾ of type ⪻special⪼ with name 'special_1'. Creation can occur with state relevant data that is present in the dataStream. Other data will get a default value.

The operation:

⪻special⪼.delete(⩽ special_1⩾)

annihilates module ⩽ special_1⩾ and removes it from the (local) storage media.

Communication

Communication occurs between compatible locations. Only modules that have equivalent type services at both ends can communicate in a modular fashion. Not the modules, but the corresponding data are communicated. Data are sent in a compressed and encrypted way. All modules that contain storable data, may take part in the communication process. Only the data that are relevant to the state of the encapsulating module will be retrieved and become part of the communication process. On the receiving side a new, module or modular system is generated, where the transmitted data are used to initialize the relevant state of the receiving system. The receiving system has a creation method that initializes new submodules with the transferred state relevant data. Communication services are linked with the necessary storage and style definition services.

System coupling

At each location a governance service couples and decouples modular systems. It can also generate and annihilate modular systems. In this way it supports a dynamic community.

Operation services

This paper does not yet consider operating system services such as sequencing of messaging and management of system resources. Real time operation requires an RTOS that effectively prevents dead locks and race conditions.

Discussion

The above approach makes from every relational database a modular database and from every file system a modular file system. It makes from every communication service a modular communication service and it standardizes programming such that reuse can be optimized in a global way. It will improve the robustness and reliability of software and enables to a large extent automated system configuration.

See: “Story of a War Against Software Complexity”, http://vixra.org/abs/1101.0061 and

“Managing the Software Generation Process”, http://vixra.org/abs/1101.0062 .