Core Concepts: Coupling

Mark Bools

May 29, 2021

Last Modified: 2022-02-01

Abstract

Examining the core concept of coupling, a measure of how difficult things are to separate..

When we say that something is ‘tightly coupled’ it is a criticism, not a complement. Things that are tightly coupled are difficult to separate and things that are difficult to separate are, essentially, a single thing. This becomes a critical problem when one of those things fails or changes.

Businesses often become tightly coupled to one another, to the detriment of the dependant business. Consider the business based entirely on YouTube. If YouTube change their terms of service the dependant business has no choice but to comply. If YouTube decide to ban that business, close their account, then that business is essentially ended. This is an example of the harms of tight coupling.

Software coupling, best to worst:

Message Modules communicate by calling one another but passing no data.

Data Modules communicate by passing data only.

Stamp Passing entire data structures containing ‘tramp’ data, data that is not relevant to the operation of the relationship between the two modules.

Control Modules pass control data, data that tells the recipient to behave differently.

External Modules share dependence on a module external to the software.

Common Modules share data external to both.

Content Modules share details of their internal operation.

Coupling cannot be avoided entirely, all systems need some degree of data sharing and calling between modules. The point with coupling, like cohesion, is to keep control of coupling and reduce it as much as practicable.

Tighter coupling in software often leads to ripple effects when changes are made to subordinate functions. Tight coupling can also lead to inefficient builds, in particular it is often a barrier to build parallelization as subordinate functions/modules/libraries will need to be built before their superiors.

Examples

Message

Suppose we have a module offering control over a motor in which there are two functions `start()` and `stop()`. In another module we need to start and stop the motor, so we use the two provided functions. Notice we pass no data while invoking these functions. We have created a message coupling.

Message coupling is the loosest coupling on our list. It is relatively straightforward to drop in a replacement motor module, it need only provide the `start()` and `stop()` functions.

Data

Consider if, instead of the simple `start()` and `stop()` functions we have one `speed(x)` where `x` is the desired speed. Any client calls `speed(0)` to stop the motor, but what about starting the motor? What is `x`? Can it be negative (presumably putting the motor in reverse)? Is it an integer, a float, or some other type? What does it represent, MPH, Km/h, percentage of maximum speed?

These questions need to be answered and once answered create a more complex relationship between the motor module and it's clients. To replace the motor module the new one would need to conform to the same data specification, otherwise the client will need to be changed to account for the new motor module. This is data coupling, named for the focus on data.

Stamp

Related to data coupling is stamp coupling. Data coupling becomes stamp coupling when the data passed is a composite and the components are not directly related. Sticking with the motor example, suppose we had a data structure containing various values relevant to the control and monitoring of the motor; the set speed, the motor's temperature, etc. We might decide to pass this data structure to each function in the motor module. This creates stamp coupling.

Suppose we decide to replace the motor module. The new module will likely not understand the composite data structure. It may have no need for most of the data in the structure (if the new module provides no monitoring it is unlikely to have a use for the temperature).

Stamp coupling is worse than data coupling because it not only adds a requirement that the client know something about the data used in the module but also hides other, not necessarily relevant data (adding unnecessary baggage).

Control

In control coupling the client passes some instruction to the module function that tells the function how to behave. In the motor control example, the speed function may take an optional `reverse` flag which if `true` causes the function to change the motor's speed to the reverse direction.

Control coupling is worse than stamp coupling because it can hide functionality. If we replace the motor module the new module needs to support the 'magic' `reverse` flag. Worse, suppose several modules refer to this global variable and we decide to change the units (kph to mph), it is difficult to track down all the modules that refer to this variable.

External

Instead of having a motor module, suppose we rely on the motor's own interface protocol. Any module in our system that needs to interact with the motor does so directly. This creates external coupling.

Changing the type of motor used in our system would now require us to hunt down and modify all the code in each module that used the motor control protocol, a thankless and wholly avoidable task (not to mention error prone).

Common

If, instead of passing speed as a parameter to the motor module, we keep the current motor speed in a global variable. We now have common coupling.

The problem with common coupling is, like control coupling, information is being hidden. How could a client know that they must maintain this speed variable?

Common coupling exists when modules communicate through data external to both modules. Arguably the most common and overlooked form of external coupling arises when data is put into a database.

Content

If you want to make your system monolithic, difficult to maintain and debug, and very difficult to understand then expose module's internals to one another. Content coupling is wrong, period. Any part of your system exhibiting content coupling needs to be rewritten (refactored) immediately. Content coupling is so bad it is often called 'pathological coupling'.

If our motor module held the speed in a local variable but client modules access this and set the speed. This is terrible. Like common coupling this causes problems when we want to change the meaning of the variable, but worse the motor module has a reasonable expectation that it can change internal local variables without needing to check client code.