Core Concepts: Cohesion

Mark Bools

May 28, 2021

Last Modified: 2022-02-01

Abstract

Examining the core concept of cohesion, a measure of how well things work together.

Cohesion is a measure of how well thing work together. When we say that 'X is cohesive' we mean that the things that X encapsulates all work together and can be thought of as a sensible whole.

Socially we talk of 'social cohesion' to mean that which identified groups exhibit; common purpose, standards, and behaviour. In the context of IT this is useful for teams. Team cohesion is important. Imposing team cohesion is not possible, this would be coercion no cohesion. Team cohesion must be established by the team and generally emerges naturally as the team identifies shared goals.

Social cohesion in one of the challenges in adopting cross-silo disciplines like DevOps. Silos exist in part because of team cohesion established once the team is created by some common objective. The 'operations' team are commonly tasked with 'defending the operational environment' and this common goal is often a binding force within the team. Contrary to this, poor organisations encourage project (development) teams to focus on delivery—not in itself a problem unless, as is too often the case, delivering to a specific date becomes the overriding goal. When the delivery date becomes an overriding goal this causes tension between the two teams, each will rally round their own core value resulting in a combative and uncooperative environment.

Entering into such an environment with the goal of breaking down the silos and building a DevOps culture means redefining these goals, breaking team cohesion by changing the objectives that bind the teams together and recreating team cohesion such that they focus on a common goal. Of course one needs to be wary not to create a new cohesion problem, IT versus the business. There are many difficulties with this movement.

Efforts to impose DevOps often result in massive teams. Wrong! Firstly, large teams naturally have less cohesion and we want higher cohesion. Secondly, DevOpsis not about creating *the* DevOps team but creating *a* DevOps culture.

System cohesion, best to worst:

Functional All members of the module operate to support one common well-defined operation with no side effects.

Sequential Members of the module process data between one another. A client of the module is expected to preserve and pass the data sequentially between members (think 'production line').

Cummunicational All members operate exclusively to maintain one data structure.

Procedural Members are all part of an algorithm with a specific sequence.

Temporal Members are related only by the time during which they operate. For example, a module that contains all functionality for a boot sequence, or a module containing all functionality for shutdown.

Logical Members perform similar operations. For example, a module containing all 'input' functions, or a module containing all error handling.

Coincidental The module is a bag of loosely related functions. Modules containing 'all the stuff we couldn't find a home for' are classic coincidentally cohesive modules that show a lack of forethought or design.

Notice that replacing 'module' with 'organisation' or 'team' makes much of this relevant to your organisation and team too.

The degree of cohesion is judged from the perspective of the module's clients. While high cohesion is preferred because modules with higher cohesion tend to be easier to use, maintain, and replace in a system, care must be exercised that we do not make our lives difficult by trying too hard.

There is seldom one perfect 'right' answer when designing software and choosing a module's degree of cohesion is one situation where you will enjoy much debate. Choosing a *good* solution is a skill you develop over time and even then you will have debates on the best solution; this is healthy. The point being we make better software by trying to make our modules higher on the cohesion scale because more cohesive modules are easier to change.

Examples

Functional

Consider a library of arithmetic functions. Each is clearly related to the others, a client is likely to use several of the functions (add, subtract, multiply, divide, etc.), these functions have no side effects (they return the same output for given inputs and do not store any state between invocations). It is possible that the functions call one another (exponent could be implemented as repeated calls to multiplication). Above all, any client that uses one of these functions is likely to use the others. All of these features point to good functional cohesion.

Another example of good functional cohesion is a logging module (examples include `log4j` in the Java world and `logging` in Python). Logging modules provide a coherent set of functions all focussed on creating, writing, and controlling logs. Again, all of the operations are related by one well defined function (logging) and any client is likely to use a substantial subset of the module's operations.

Sequential

To implement a binary search we take the list to be searched, sort it, compare the search term with the item in the middle of the sorted list if the term is greater than the middle item then take the top half of the sorted list, find the middle, compare and repeat until the item is found or no smaller list can be created.

If we created a module offering access to the `sort` and `binarySearch` functions then then this module would have sequential cohesion as the `binarySearch` relies on `sort` being called first.

Sequential cohesion is closely related to, and confused with, procedural cohesion. The difference being that while procedurally cohesive operations all contribute to a common algorithm, sequentially cohesive operations must also chain together, one operation's output being the input to the next.

Communicational

There are many communicationally cohesive modules in many languages, libraries providing support for extended data types like hash tables for example.

An example of communicational cohesion familiar to many developers is a module grouping operations handling a website's shopping basket.

communicational

```python
def add_item(...):
  ...
def remove_item(...):
  ...
def apply_discount(...):
  ...
def calculate_total(...):
  ...
```

Communicational cohesion is not intrinsically bad but it should be a warning of potential problems. The problem is that a poor choice of data structure can result in a module that becomes a mess. Does `apply_discount` belong in the example? `calculate_total` is similarly questionable. Why might this be bad?

If we focus on the basket as simply a container for items it becomes simpler to keep the module's intent and interface clean and easier to understand, this in turn will make it much simpler to maintain. The discount and total functions are better placed into a separate module focused on invoice type function. This naturally separates the data structures; the basket module handles the list of items selected, the 'invoice' module handles 'the money'. With care these modules are freely reusable.

Procedural

Procedural cohesion is a more general form of sequential cohesion. Procedurally cohesive operations all contribute to one algorithm, the focus is on transforming module inputs into module outputs through a set of related operations. Unlike sequential cohesion, procedural cohesion does not require any specific invocation order.

Temporal

Modules that deal with system start up or shutdown are good examples of temporally cohesive modules where the contained functions are related only because they are used at the same time.

This can be tempting to inexperienced developers but consider the consequences. Let's put all our start up code in one module.

temporal

```
def init_logging():
  ...
def init_database():
  ...
def init_user_interface():
  ...
```

These operations have nothing in common other than being used as the application starts up. What is the problem? Suppose we want to reuse our database code in another project. There is no neat way to take the initialisation code from this system without also taking logging and user interface code, or extracting (copying) the database code into the new system (into a new start up module).

Logical

Barely better than coincidental cohesion, logically cohesive operations share a purpose ('handle all input') but little else.

Say we have a management unit for a car and put all the code to handle reading sensors into one module. This module is 'logical' in the sense that all the operations 'read sensors' but this is bad because the subsequent processing of this data will be handled elsewhere.

```
logical
def read_wheel_speed():
  ...
def read_engine_speed():
  ...
def read_speedometer_position():
  ...
```

Suppose we change the speedometer used, now we change `read_speedometer_position()` but there will be other operations spread throughout our system (like an `update_speedometer_position()` in a 'handle all output' module). Basically a horror show.

Coincidental

A classic, and widely used, example of a coincidentally cohesive module is the C `stdlib`. What do `malloc` (allocate block of memory) and `rand` (generate a random number) have in common? Basically nothing, no shared data, and no common purpose or function. Yet both are members of `stdlib`. These functions are coincidentally related, the only thing they have in common is that they are members of this module.