CM2010: Software Design and Development

Summary

Arjun Muralidharan

8th November 2020

Contents

List of Figures

List of Listings

1 Modules and module complexity

Learning Outcomes

✓ Assign different categories of module coupling and cohesion to a given program

✓ Write programs using variables, control flow and functions

Important reference points for software desing and development are the Software Engineering Body of Knowledge (SWEBOK) and IEEE vocabulary.

1.1 Module Complexity

Modularity A mechanism for improving the **flexibility** and **comprehensebility** of a system while allowing the **shortening** of its development time.

Module A program unit that is discrete and identifiable with respect to **compiling**, **combining** and **loading**. It is a **logically separable** part of a program. It can be represented by a **set of source code files** under version control that can be manipulated together as one. It is a collection of both **data and routines** that act on it (such as a class).

Complexity The degree to which a system's design or code is **difficult to understand** because of **numerous components** or relationships among components, any of a set of structure-based metrics that measure these attributes, or the degree to which a system or component has a design or implementation that is difficult to understand and verify

Simplicity The degree to which a system or component has a design and implementation that is straightforward and easy to understand, or software attributes that provide implemenetation of functions in the **most understandable manner**.

Classical metrics Software complexity can be measured using the **cyclometric complexity**, that is the number of execution paths through the code, **coupling**, that is how much modules interact with each other, and **cohesion**, that is the amount of functionality in a single module..

Fat and Tangle The terms *fat and tangle* refer to the amoung of cohesion and content of a module (*fat*) and the amount of interaction found between modules (*tangle*). Breaking up a program into modules reduces *fat* but increases **tangle**. Issues arise if the tangle is cyclical, that is modules interact with other modules in a circular dependency.

1.1.1 Measures of Complexity

Scientific literature describes multiple approaches to measuring complexity.

McCabe [1]describes **cyclomatic complexity** as a measure based on graph theory. This approach measures the number of unique paths taken through a program calculated as the cyclomatic complexity $v(G)$:

$$v(G) = e - n + 2p$$

for a graph G with e edges, n vertices and p connected components (subgraphs where any two nodes are connected by a path). McCabe then assigned a cyclomatic number to various common control structures such as sequences, conditional statements, and loops as shown in Figure 1. Properties of the cyclomatic number are:

1. $v(G) \geq 1$
2. $v(G)$ is the maximum number of linerarly independent paths in G.
3. Inserting or deleting functional statement to G does not affect $v(G)$.
4. G has only one path if and only if $v(G) = 1$.
5. Inserting a new edge in G increases $v(G)$ by 1.
6. $v(G)$ depends only on the decision structure of G.

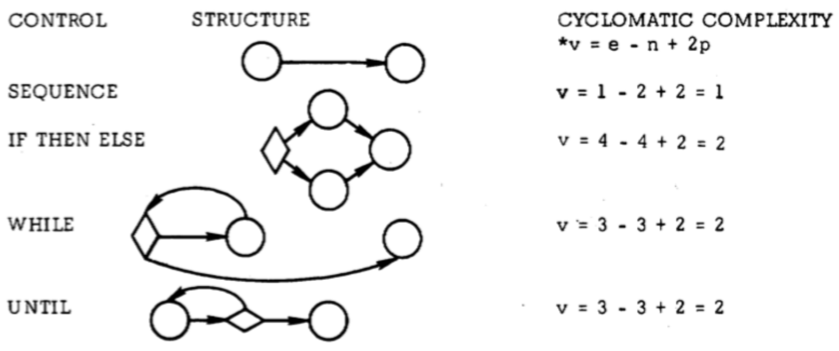

Figure 1. *caption*

Sangwan [2] measured **excessive complexity** based on Structure 101 over time as "structural epochs". This method measures complexity in *fat* and *tangles*, with fat representing the difficulty in understanding a specific module, and tangle representing the number of cyclical dependencies between modules. The method removes as many edges as possible to achieve as close to a *directed acyclic graph* as possible.

Sangwan found in a review of open source projects that excessive complexity shifts upward from lower level modules to higher level modules, and often grows over time until a refactoring occurs, reducing the excessive complexity to an idealized growth line, shown in Figure 2.

Other approaches define **scenario-based** metrics that measure how well a system support specific business scenarios (such as adapting the system to a business change or support for legacy systems).

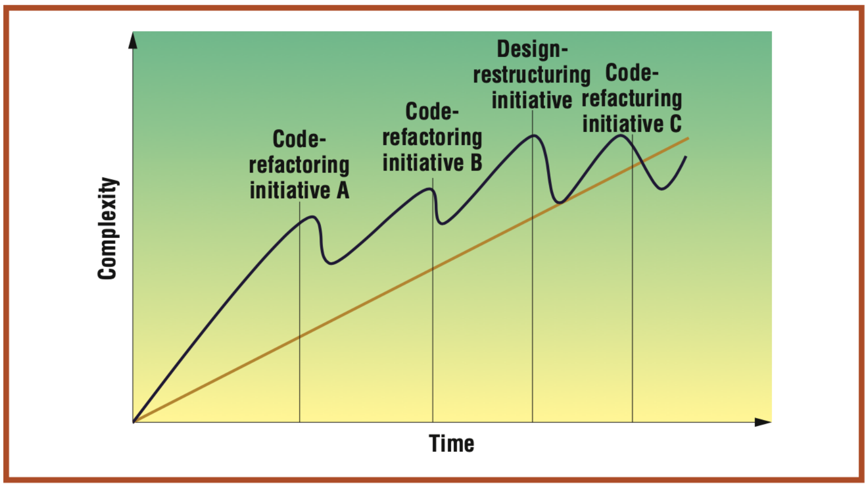

Figure 1. Idealized evolutionary pattern in which complexity grows until code refactoring, design restructuring, or architectural renovation is necessary. (The gold line represents idealized growth of source lines of code over time.)

Figure 2. *caption*

1.2 Module Cohesion

Cohesion is the manner and degree to which the tasks performed by a **single software module** are **related to one another** within a single module. It is the **measure of the strength of association** of the elements in a module.

There are 7 types of cohesion.

Communicational Cohesion The tasks performed by a module use the **same input data** or contribute to producing the **same output data**. This kind of cohesion is **always** acceptable.

Functional Cohesion The tasks performed by a module all **contribute to the performance of a single function**. This kind of cohesion is **always** acceptable. It is also the foundation of object-oriented programming.

Logical Cohesion The tasks performed by a module perform **logically similar functions**. This kind of cohesion can **sometimes** be acceptable but generally considered bad as software that might *look* similar but do completely different things.

Procedural Cohesion The tasks performed by a module all contribute to a **given program procedure** such as an iteration or decision process. This kind of cohesion is **rarely** considered good, as a module works on different data in each step of a procedure.

Sequential Cohesion The **output of one task** performed by a module serves as the **input to another task** performed by the module. This kind of cohesion is **never** acceptable as it combines parts of the program that happen in sequence, but might do completely different things with completely different data.

Temporal Cohesion The tasks performed by a module are all **required at a particular phase of program execution**. This kind of cohesion is **never** acceptable, as it combines program parts that happen at the same time but otherwise migth do completely different things, making it hard to understand the individual parts.

Coincidental Cohesion The tasks performed by a module have **no functional relationship** to each other. This kind of cohesion is **never** acceptable as it's completely random.

1.3 Module Coupling

Coupling is the manner and degree of interdependence between modules, the strength of the relationship between modules or how closely related two routines or modules are. There are types of coupling.

Common-environment coupling Type of coupling in which two software modules access a common data area. This is **acceptable** as long as modules are not operating on global data, which might result in unexpected behaviour. Modules should be limited in their access to a specific data area, possibly in sub-environments.

Content Coupling Type of coupling in which some or all of the contents of one software module are included in the contents of another module. An example of this are **Lambda functions** in C++ or Python, or event listeners. This is **acceptable** as long as the contained module really remains contained and not accessed from outside the parent module directly by building bridges into the submodule.

Control coupling Type of coupling in which one software module communicates information to another module for the explicit purpose of influencing the latter module's execution. This is considered **bad** because the latter module can become quite complicated. It is better to move the logic of decision making into the first module and call separate, distinct, clearly understood subroutines from it.

Data Coupling Type of coupling where output from one module serves as input to another module. This is the most common use in the form of functions that have a single purpose and receive some data from another part of the program. This is **acceptable** as long as the receiving function does the same thing with any input data (and does not morph into control coupling).

Hybrid coupling Type of coupling in which different subsets of the range of values that a data item can assume are used for different and unrelated purposes in a different software module. For example, two modules might use the same data in completely different ways. Module 1 might use some integer value to store a timestamp, while another uses it to calculate the color of a pixel. This was common when memory was limited, and considered **bad**.

Pathological Coupling Type of coupling in which one module affects or depends upon the internal implementation of another. This is **bad** as it can result in very unpredictable behaviour, which is not desirable outside of perhaps performative programming.

2 Test-driven development

Learning Outcomes

✓ Define test driven development and write unit tests

✓ Write programs using variables, control flow and functions

2.1 Definition of TDD

Test-driven development is the discipline of writing tests first and working code to pass those tests. Uncle Bob (Robert C. Martin) outlines the three laws of TDD [3]:

1. You may not write production code unless you've first written a failing unit test.
2. You may not write more of a unit test than is sufficient to fail.
3. You may not write more production code than is sufficient to make the failing unit test pass.

This loop is supposed to occur within 2 minutes, making the development process very stable, resulting in your code being working most of the time.

Unit tests can be of different types:

Interface Testing Programmatic interfaces (not graphical interfaces) can be tested by evaluating that a module receives certain inputs and provides certain outputs. This might including the types of inputs, including how default arguments of a function are handled, and the outputs.

Exercising data structures Verifying data structure and their correct usage. This might include the number of items in a data structure, ensuring a data structure retains integrity and what might happen at the limits of the data structure (e.g. if it is too large).

Boundary testing We can evaluate what happens when we pass certain inputs at the boundaries of acceptable inputs? This kidn of test makes sure that the code works correctly when handling data at boundaries, e.g. have we looked at the beginning and end of an array properly?

Execution Paths Tests that follow all deliberate paths of the module execution ensure that the code works in all possible conditions by sending it different kinds of arguments to explore different pathways. This kind of test would also indicate if **control coupling** is too high in the given module.

3 Robust and secure programming

Learning Outcomes

- ✓ Use defensive coding and exception handling techniques to prevent processing of invalid data and to handle unexpected events
- ✓ Write programs using variables, control flow and functions

3.1

4 User testing

Learning Outcomes

- ✓ Describe how user testing can be carried out and evaluated

4.1

5 Version control

Learning Outcomes

- ✓ Use version control tools to manage a codebase individually and collaboratively

5.1

References

[1] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976. [Online]. Available: https://ieeexplore.ieee.org/document/1702388

[2] R. S. Sangwan, P. Vercellone-Smith, and P. A. Laplante, "Structural epochs in the complexity of software over time," *IEEE Software*, vol. 25, no. 4, pp. 66–73, 2008. [Online]. Available: https://ieeexplore.ieee.org/document/4548410

[3] R. C. Martin, "Professionalism and test-driven development," *IEEE Software*, vol. 24, no. 3, pp. 32–36, 2007. [Online]. Available: https://ieeexplore.ieee.org/document/4163026