Software Design And Development (CM2010)

Course Notes

Felipe Balbi

October 14, 2020

Contents

Week 1

Key Concepts

- Define the terms module and module complexity in terms of computer programs and systems.
- Identify the modules present in computer programs and systems.
- Analyse program code in terms of its complexity.

Module Introduction

The objectives of the module are:

1. Write programs using control flow, variables, and functions
2. Use defensive coding and exception handling techniques to prevent processing of invalid data and to handle unexpected events
3. Use Version Control tools to manage a codebase individually and collaboratively (`git`)
4. Define Test-Driven Development and write Unit Tests
5. Assign different categories of module coupling and cohesion to a given program
6. Describe how User Testing can be carried out and evaluated

We will use three different languages throughout the course. They are: C++, Python, and JavaScript.

1.0202 Reference points

Two main references will be used during this course: the *SWEBOK* and ISO/IEC/IEEE 24765:2010 - IEEE Systems And Software Engineering Vocabulary.

SWEBOK is the Software Engineering Body Of Knowledge. From this reference material, we focus on the topic of *Design*.

Design is concerned with the Design fundamentals, key issues of software, software structure and architecture, user interface design, software design quality analysis and evaluation, software design notations, software design strategies and methods, and software design tools.

ISO/IEC/IEEE 24765:2010 is a sort of *dictionary* defining common terms.

1.0203 SWEBOK guide and IEEE vocab

- ISO/IEC/IEEE International standard – Systems and software engineering – Vocabulary, ISO/IEC/IEEE 24765:2010(E) Dec 2010, pp.1–418.
- R.E.D. Fairley, P. Bourque and J. Keppler, The impact of SWEBOK Version 3 on software engineering education and training in 2014 IEEE 27th Conference on Software Engineering Education and Training (CSEE&T). (Klagenfurt, Austria: IEEE, 2014).

1.0204 What is a module?

"... a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time". Parnas, 1972.

Once software has been modularized, different parts can be replaced and/or used in different software. Moreover, modularity makes software easier to understand because each small piece can be studied and understood in isolation.

During this course, we define a module as:

- program unit that is discrete and identifiable with respect to **compiling**, **combining** with other units, and **loading**;
- logically separable part of a program;
- set of source code files under version control that can be manipulated as one;
- collection of both data and the routines that act on it.

1.0206 What is module complexity?

Complexity is defined as:

1. The degree to which a system's design or code is **difficult to understand** because of numerous components or relationships among components;
2. Pertaining to any of a set of structure-based **metrics** that measures the attributes in (1);
3. The degree to which a system or component has a design or implementation that is difficult to understand and **verify**.

Simplicity is defined as:

1. The degree to which a system or component has a design or implementation that is **straightforward** and **easy** to understand;
2. Software attributes that provide implementation of functions in the most understandable manner.

1.0208 Complexity references

Please read the following articles. The first is a paper about structural complexity and how it changes over time. The second is the classic McCabe paper on module complexity.

- R.S. Sangwan, P. Vercellone-Smith and P.A. Laplante 'Structural epochs in the complexity of software over time', IEEE Software 25(4) Jul-Aug 2008, pp.66–73.
- T.J. McCabe 'A complexity measure', IEEE Transactions on Software Engineering SE-2(4) Dec 1976, pp.308–320.

Week 2

Key Concepts

- Define module cohesion in terms of computer program architecture.
- Define types of module cohesion and identify them in computer programs.
- Use programming techniques to improve module cohesion.

2.0102 Week 2 reading

This document contains the definitions of various types of module cohesion that you will encounter in the videos. We recommend that you download this and keep it to hand while you watch the videos.

ISO/IEC/IEEE International standard – Systems and software engineering – Vocabulary, ISO/IEC/IEEE 24765:2010(E) Dec 2010, pp.1–418.

2.0201 What is module cohesion?

Module cohesion is a way to reason about the contents of a module.

We're concerned about a single module and its contents, not about interactions between modules.

From the ISO Standard Software Engineering Vocabulary, Module Cohesion is defined as:

- Manner and degree to which the tasks performed by a single software module are related to one another;
- In software design, a measure of the strength of association of the elements within a module.

What these tell us is that the *stuff* within a module should be strongly related, otherwise, perhaps, they shouldn't be part of a single module. In summary, a module should do a single thing and a single thing only.

There are several types of module cohesion, they are:

Communicational Type of cohesion in which the tasks performed by a software module use the same input data or contribute to producing the same output data

Functional Type of cohesion in which the tasks performed by a software module all contribute to the performance of a single function

Logical Type of cohesion in which the tasks performed by a software module perform logically similar functions

Procedural Type of cohesion in which the tasks performed by a software all contribute to a given program procedure such as an iteration or decision process

Sequential Type of cohesion in which the output of one task performed by a software module serves as input to another task performed by the module

Temporal Type of cohesion in which the tasks performed by a software module are all requried at a particular phase of program execution

Coincidental Type of cohesion in which the tasks performed by a software module have no functional relationship to one another

2.0204 Why are different types of module cohesion good or bad?

Communicational cohesion is **good** because it's about combining things in a single module that are working on similar data.

Functional cohesion is **good** too. That's because we put into a module functions that work together to achieve a certain goal.

Logical cohesion is generally considered **bad**. Just because software looks like it's doing similar things, doesn't necessarily mean they should be part of the same module.

Procedural cohesion is **bad**. It tends to result in large procedures that do many communicationally different things.

Sequential cohesion is **bad**. This is the idea that a program is doing a sequence of things and, as such, that sequence of things should be put together.

Temporal cohesion is **bad**. This is the idea that because things are happening at the same time, they should be put together.

Coincidental cohesion is **bad**. In fact, this is terrible. Things are put together due to mere coincidence. They just happen to be placed together.

Week 3

Key Concepts

- Give a high-level definition of module coupling and illustrate with an example.
- Analyse computer programs to identify different types of module coupling.
- Describe different types of module coupling and discuss which are desirable and which are not.

3.0102 Week 3 reading

ISO/IEC/IEEE International standard – Systems and software engineering – Vocabulary, ISO/IEC/IEEE 24765:2010(E) Dec 2010, pp.1–418.

3.0201 What is module coupling?

Module Coupling is defined as:

- Manner and degree of interdependence between software modules
- Strength of the relationships between modules
- Measure of how closely connected two routines or modules are
- In software design, a measure of the interdependence among modules in a computer program

3.0203 Different types of coupling: good or bad?

There are different types of module coupling:

Common Environment type of coupling in which two software modules access a common data area

Content type of coupling in which some or all of the contents of one software module are included in the contents of another module

Control type of coupling in which one software module communicates information to another module for the explicit purpose of influencing the latter module's execution

Data type of coupling in which output from one module serves as input to another module

Hybrid 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 different software modules

Pathological type of coupling in which one software module affects or depends upon the internal implementation of another

3.0205 Common environment coupling: good or bad?

Common environment coupling is where two modules have a shared environment. In this environment we have two modules and a block data.

Both modules have access to the data and can change it and do what they like to it as it's shared.

While this sounds like a good idea, it can result in *Race Conditions* where one module changes the data without the other module knowing about it. A better approach would be to have smaller environments of modules each with their own data.

Because of this reason, Common Environment Coupling is not necessarily bad, but one must be careful when implementing it in order to limit its scope.

3.0207 Content coupling: good or bad?

Content coupling would be like having one module (*m1*) with another module (*m2*) contained inside it. It makes it so that nothing outside of *m1* can see *m2* or even know it exists.

It is a regular type of module coupling which is used, for example, is event listeners in JavaScript.

3.0209 Control coupling: good or bad?

Control coupling is when one module is modifying the operation of another one by sending a flag to change how it operates.

Because of that, module 1 tends to become rather complicated. It's easier to illustrate with some python-like code:

```
1 def compute(a, b, op):
2     if op == "add":
3         return a + b
4     else if op == "mul":
5         return a * b
6     else if op == "div":
7         return a / b
```

```
8       else if op == "sub":
9           return a - b
```

While the above is a very contrived example, it's already clear how cumbersome this is. It would have been better to split compute into `add`, `mul`, `div`, and `sub` primitives.

3.0211 Data coupling: good or bad?

Data or Input/Output coupling looks like a production line. The output of one module is fed as input into another. Basically, we can look at it as a function call. Module 1 calls module 2 to run some computation. Data coupling is good.

3.0213 Hybrid coupling: good or bad?

Hybrid coupling is when different modules treat the data source in different ways. In general, this can get confusing and things can go awry very easily.

An example may be that we use a single integer, e.g. 32-bits, and subdivide it into something of our own. Perhaps bit 31 is some Dirty/Clean flag, bits 28:30 hold a state of a state machine, bits 22:27 hold some size information to whatever other memory area and the remaining bits 21:0 hold the top-most 22 bits of the address to a memory area. It should be clear that this can get confusing.

There are cases where this sort of memory usage is the only option, but in general Hybrid Coupling is bad.

3.0215 Pathological coupling: good or bad?

"It's a bit like control coupling, but worse"[1]. Module 1 can either affect the internal workings module 2, or it has a direct dependency in its implementation. This means that module 2 can't be easily replaced, refactored, modified. It's bad.

[1] MYK has awesome comments :-)