Modularity and Coupling and Cohesion

- Sources used in preparing these slides:
  - Lecture notes by Sagar Naik and Mattias Hembruch
  - Object-Oriented and Classical Software Engineering by Stephen Schach (McGrowHill, 2002)
  - Lecture notes by Paul Dasiewicz

Overview

➔Design Quality

- Modularity
- Coupling and Cohesion

Design Quality

- Obviously design should correctly implement a specification
  - Both functional and non-functional requirements
- But also:
  - Enable efficient code production & division of work among team members (basis for project management)
  - Be minimal (as simple as possible while still addressing all requirements)
  - Be clear and understandable (allow intellectual control over a project and manage its complexity), maintainable and extensible, …
- Need to modularize design in order to manage complexity

Overview

- Design Quality

➔Modularity

- Coupling and Cohesion

What is a module?

- Common view: a piece of code. Too limited.
- Compilation unit, including related declarations and interface
- David Parnas: a unit of work.

➔Collection of programming units (procedures, classes, etc.)

  – with a well-defined interface and purpose within the entire system,
  – that can be independently assigned to a developer

Why modularize a system?

- **Management**: Partition the overall development effort
  - Divide and conquer (actually: “Divide et impera” = “Divide and rule”)
- **Evolution**: Decouple parts of a system so that changes to one part are isolated from changes to other parts
  - Principle of directness (clear allocation of requirements to modules, ideally one requirement (or more) maps to one module)
  - Principle of continuity/locality (small change in requirements triggers a change to one module only)
- **Understanding**: Permit system to be understood
  - as composition of mind-sized chunks, e.g., the 7±2 Rule
  - with one issue at a time, e.g., principles of locality, encapsulation, separation of concerns
- Key issue: what criteria to use for modularization?

Information hiding (Parnas)

- Hide secrets. OK, what's a "secret"?
  - Representation of data
  - Properties of a device, other than required properties
  - Implementation of world models
  - Mechanisms that support policies
- Try to localize future change
  - Hide system details likely to change independently
  - Separate parts that are likely to have a different rate of change
  - Expose in interfaces assumptions unlikely to change

What Is an Interface?

- Interface as a contract - whatever is published by a module that
  - *Provided interface*: clients of the module can depend on and
  - *Required interface*: the module can depend on from other modules
- Syntactic interfaces
  - How to call operations
    - List of operation signatures
    - Sometimes also valid orders of calling operations
- Semantic interfaces
  - What the operations do, e.g.,
    - Pre- and post-conditions
    - Use cases
  - Performance specification
  - Reliability specification
  - ...

Further Principles

- Explicit interfaces
  - Make all dependencies between modules explicit (no hidden coupling)
- Low coupling - few interfaces
  - Minimize the amount of dependencies between modules
- Small interfaces
  - Keep the interfaces narrow
    - Combine many parameters into structs/objects
    - Divide large interfaces into several interfaces
- High cohesion
  - A module should encapsulate some well-defined, coherent piece of functionality (more on that later)

Overview

- Design Quality
- Modularity

➔Coupling and Cohesion

Coupling and Cohesion

- Cohesion is a measure of the coherence of a module amongst the pieces of that module.
- Coupling is the degree of interaction between modules.
- You want *high cohesion* and *low coupling*.

Degrees of Cohesion

1. Coincidental cohesion
2. Logical cohesion
3. Temporal cohesion
4. Procedural cohesion
5. Communicational cohesion
6. Functional cohesion
7. Informational cohesion

Low cohesion - bad

High cohesion - good

Coincidental cohesion

- The result of *randomly* breaking the project into modules to gain the benefits of having multiple smaller files/modules to work on
  - Inflexible enforcement of rules such as: “every function/module shall be between 40 and 80 lines in length” can result in coincidental coherence
- Usually worse than no modularization
  - Confuses the reader that may infer dependencies that are not there

Logical cohesion

- A “template” implementation of a number of quite different operations that share some basic course of action
  - variation is achieved through parameters
  - “logic” - here: the internal workings of a module
- Problems:
  - Results in hard to understand modules with complicated logic
  - Undesirable coupling between operations
- Usually should be refactored to separate the different operations

Example of Logical Cohesion

```
void function(param1, param2, param3, ..., paramN)
{
    variable declarations....
    code common to all cases... [A]
    if ( param1 == 1 ) [B]
            ...
    else if ( param1 == 2 )
            ...
    else if ( param1 == n )
            ...
    end if
    code common to all cases... [C]
    if ( param == 1) [D]
            ...
    else if ( param1 == 5 )
            ...
    end if
    code common to all cases... [E]
    if ( param1 == 7 )
            ...
}
```

Temporal Cohesion

- Temporal cohesion concerns a module organized to contain all those operations which occur at a similar point in time.
- Consider a product performing the following major steps:
  - Initialization, get user input, run calculations, perform appropriate output, cleanup.
- Temporal cohesion would lead to five modules named initialize, input, calculate, output and cleanup.
- This division will most probably lead to code duplication across the modules, e.g.,
  - Each module may have code that manipulates one of the major data structures used in the program.

Procedural Cohesion

- A module has procedural cohesion if all the operations it performs are related to a sequence of steps performed in the program.
- For example, if one of the sequence of operations in the program was “read input from the keyboard, validate it, and store the answers in global variables”, that would be procedural cohesion.
- Procedural cohesion is essentially temporal cohesion with the added restriction that all the parts of the module correspond to a related action sequence in the program.
- It also leads to code duplication in a similar way.

Procedural Cohesion

Module A

```
operationA()
{ readData(data,filename1);
  processAData(data);
  storeData(data,filename2);
}

readData(data,filename)
{ f := openfile(filename);
  readrecords(f, data);
  closefile(f);
}
storeData(data,filename)
{...}
processAData(data)
{...}
```

Module B

```
operationB()
{ readData(data,filename1);
  processBData(data);
  storeData(data,filename2);
}

readData(data,filename)
{ f := openfile(filename);
  readrecords(f, data);
  closefile(f);
}
storeData(data,filename)
{...}
processBData(data)
{...}
```

Communicational Cohesion

- Communicational cohesion occurs when a module performs operations related to a sequence of steps performed in the program (see procedural cohesion) AND all the actions performed by the module are performed on the same data.
- Communicational cohesion is an improvement on procedural cohesion because all the operations are performed on the same data.

Functional Cohesion

- Module with functional cohesion focuses on exactly one goal or “function”
  - (In the sense of purpose, not a programming language “function”).
- Module performing a well-defined operation is more reusable, e.g.,
  - Modules such as: read_file, or draw_graph are more likely to be applicable to another project than one called initialize_data.
- Another advantage of is fault isolation, e.g.,
  - If the data is not being read from the file correctly, there is a good chance the error lies in the read_file module/function.

Informational Cohesion

- Informational cohesion describes a module as performing a number of actions, each with a unique entry point, independent code for each action, and all operations are performed on the same data.
  - In informational cohesion, each function in a module can perform exactly one action.
- It corresponds to the definition of an ADT (abstract data type) or object in an object-oriented language.
- Thus, the object-oriented approach naturally produces designs with informational cohesion.
  - Each object is generally defined in its own source file/module, and all the data definitions and member functions of that object are defined inside that source file (or perhaps one other source file, in the case of a .hpp/.cpp combination).

Levels of Coupling

5. Content Coupling (High Coupling - Bad)
4. Common Coupling
3. Control Coupling
2. Stamp Coupling
1. Data Coupling (Low Coupling - Good)

(Remember: no coupling is best!)

Content Coupling

- One module directly refers to the content of the other
  - module 1 modifies a statement of module 2
    - Assembly languages typically supported this, but not high-level languages
    - COBOL, at one time, had a verb called alter which could also create self-modifying code (it could directly change an instruction of some module).
  - module 1 refers to local data of module 2 in terms of some kind of offset into the start of module 2.
    - This is not a case of knowing the offset of an array entry - this is a direct offset from the start of module 2's data or code section.
  - module 1 branches to a local label contained in module 2.
    - This is not the same as calling a function inside module 2 - this is a goto to a label contained somewhere inside module 2.

Common Coupling

- Common coupling exists when two or more modules have read *and* write access to the same global data.
- Common coupling is problematic in several areas of design/maintenance.
  - Code becomes hard to understand - need to know all places in all modules where a global variable gets modified
  - Hampered reusability because of hidden dependencies through global variables
  - Possible security breaches (an unauthorized access to a global variable with sensitive information)
- It’s ok if just one module is writing the global data and all other modules have read-only access to it.

Common Coupling

- Consider the following code fragment:

```
while( global_variable > 0 )
{ switch( global_variable )
   {  case 1: function_a(); break;
      case 2: function_b(); break;
      ...
      case n: ...
   }
   global_variable++;
}
```

Common Coupling

- If function_a(), function_b(), etc can modify the value of global variable, then it can be extremely difficult to track the execution of this loop.
- If they are located in two or more different modules, it becomes even more difficult
  - potentially all modules of the program have to be searched for references to global variable, if a change or correction is to take place.
- Another scenario is if all modules in a program have access to a common database in both read and write mode, even if write mode is not required in all cases.
- Sometimes necessary, if a lot of data has to be supplied to each module

Control Coupling

- Two modules are control-coupled if module 1 can directly affect the execution of module 2, e.g.,
  - module 1 passes a “control parameter” to module 2 with logical cohesion, or
  - the return code from a module 2 indicates NOT ONLY success or failure, but also implies some action to be taken on the part of the calling module 1 (such as writing an error message in the case of failure).
- The biggest problem is in the area of code re-use: the two modules are not independent if they are control coupled.

Stamp Coupling

- It is a case of passing more than the required data values into a module, e.g.,
  - Passing an entire employee record into a function that prints a mailing label for that employee. (The data fields required to print the mailing label are name and address. There is no need for the salary, SIN number, etc.)
- Making the module depend on the names of data fields in the employee record hinders portability.
  - If instead, the four or five values needed are passed in as parameters, this module can probably become quite reusable for other projects.
- As with common coupling, leaving too much information exposed can be dangerous.

Data Coupling

- Data coupling exhibits the properties that all parameters to a module are either simple data types, or in the case of a record being passed as a parameter, all data members of that record are used/required by the module. That is, no extra information is passed to a module at any time.