CS189A - Capstone

Christopher Kruegel
Department of Computer Science
UC Santa Barbara
http://www.cs.ucsb.edu/~chris/

Fundamental Design Principles

There are some fundamental principles in software engineering:

- **Anticipation of Change**
  - We talked about this a lot in the context of software process models. The main principle behind agile software development.
- **Separation of Concerns**
  - You can see the use of this principle in the requirements analysis and specification. For example: separating functional requirements from performance requirements.
- **Iterative (Stepwise) Refinement**
  - For example, separating architectural design from detailed design
- **Modularity**
  - This is what I will talk about today as it applies to software design
- **Abstraction**
  - We will see examples of this when we discuss design patterns

Software Design

- We can think of software design in two main phases
  - Architectural Design
    - Divide the system into a set of modules
    - Decide the interfaces of the modules
    - Figure out the interactions among different modules
  - Detailed Design
    - Detailed design for individual modules
    - Write the pre and post-conditions for the operations in each module
    - Write pseudo code for individual modules explaining key functionality

Modularity

- Modularity principle suggests dividing a complex system into simpler pieces, called modules
- When we have a set of modules, we can use separation of concerns and work on each module separately
- Modularity can also help us to create an abstraction of a module's environment using interfaces of other modules

Modularization

- According to Parnas
  - *"... modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time."*

- The goals of modularization are to
  - reduce the complexity of the software
  - and to improve
    - maintainability
    - reusability
    - productivity

Benefits of Modularization

- Managerial (productivity)
  - development time should be shortened because several groups work on different modules with limited need for communication
- Product flexibility (reusability, maintainability)
  - it should be possible to make changes to one module without the need to change others
- Comprehensibility (reducing complexity)
  - it should be possible to study the system one module at a time

Modularization

- Gouthier and Pont:

  "*A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently ... Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching*."

Modularization

- A module is a responsibility assignment rather than a subprogram
- Question: What are the criteria to be used in dividing the system into modules?

Modularization

- In dividing a system into modules we need some guiding principles.
  - What is good for a module?
  - What is bad for a module?

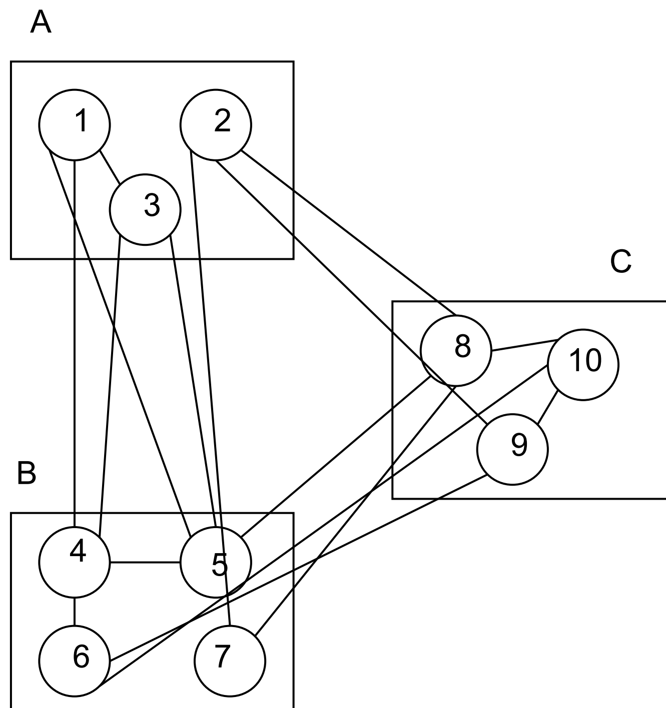
- There are two notions which characterize good things and bad things about modules nicely
  - Cohesion
    - We want highly cohesive modules
  - Coupling
    - We want low coupling between modules

Cohesion and Coupling

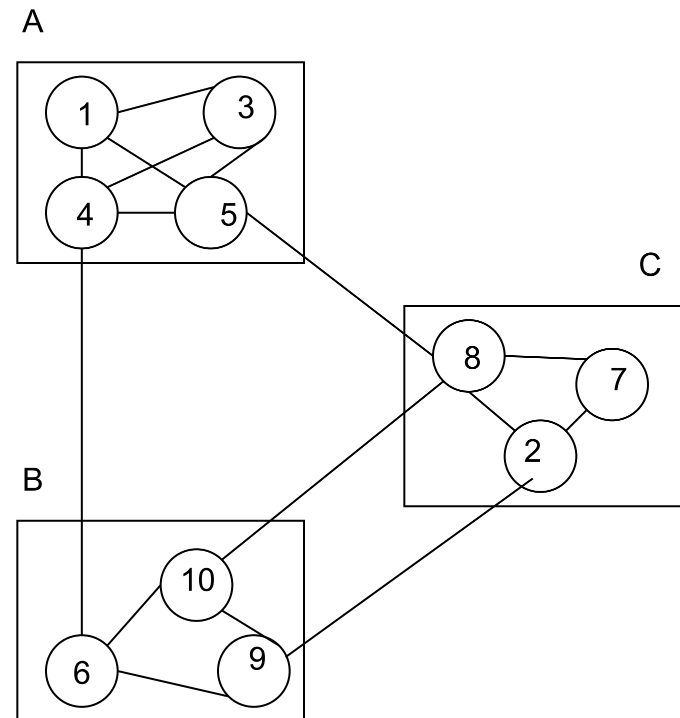
- What is cohesion?
  - Type of association among different components of a module
  - Cohesion assesses why the components are grouped together in a module

- What is Coupling?
  - A measure of strength of interconnection (the communication bandwidth, the dependencies) between modules
  - Coupling assesses the kind and the quantity of interconnections among modules

Cohesion and Coupling



Bad modularization:
low cohesion, high coupling



Good modularization:
high cohesion, low coupling

Cohesion and Coupling

- Good modularization: high cohesion and low coupling

- One study on Fortran routines found that 50% of highly cohesive units were fault free, whereas only 18 percent of routines with low cohesion were fault free [Card, Church, Agesti 1986]

- Another study found that routines with the highest coupling to cohesion ratios had 7 times as many errors as those with the lowest coupling to cohesion ratios and were 20 times as costly to fix [Selby and Basili 1991]

Types of Cohesion

- There are various informal categorizations of cohesion types in a module. I will discuss some of them (starting with the ones which are considered low cohesion)

- WORST: Coincidental cohesion
  - Different components are thrown into a module without any justification, i.e., they have no relation to each other
    - Maybe this was the last module where all the remaining components were put into
  - Obviously, this type of cohesion is not good! It basically corresponds to lack of cohesion.

Types of Cohesion

- BAD: Logical cohesion
  - A module performs multiple somewhat related operations one of which is selected by a control flag that is passed to the module
  - It is called logical cohesion because the control flow (i.e. the “logic”) of the module is the only thing that ties the operations in the module together

```
procedure operations (data1, data2, operation)
{
  switch (operation) {
    case ...: // execute operation 1 on data1
    case ...: // execute operation 2 on data2
  }
}
```

Types of Cohesion

- BAD: Temporal cohesion
  - A module performs a set of functions related in time
    - For example an initialization module performs operations that are only related by time
  - These operations can be working on different data types
  - A user of a module with temporal cohesion can not call different operations separately

```
procedure initialize_game()
{
    // initialize the game board
    // set players' scores to 0
}
```

Types of Cohesion

- Coincidental, logical and temporal cohesion should be avoided.
- Such modules are hard to debug and modify.
- Their interfaces are difficult to understand.

Types of Cohesion

- BETTER: Communicational cohesion
  - Grouping a sequence of operations that operate on the same data in the same module
  - Some drawbacks: Users of the module may want to use a subset of the operations.

```
procedure operations1and2 (data)
{
    // execute operation 1 on data
    // execute operation 2 on data
}
```

Types of Cohesion

- GOOD: Functional cohesion
  - Every component within the module contributes to performing a single function
  - Before object orientation this was the recommended approach to modularization.
  - No encapsulation between a data type and operations on that data type

```
procedure operation1 (data)
{
    // execute operation 1 on data
}

procedure operation2 (data)
    // execute operation 2 on data
}
```

Types of Cohesion

- BEST: Informational Cohesion
  - This term is made up to mean the data and functionality encapsulation used in object oriented design

```
module stack
// definition of the stack data type
procedure initialize() { .. }
procedure pop() { .. }
procedure push() { .. }
procedure top_element() { .. }
```

- A ranking of (from good to bad) types of cohesion:

informational > functional > communicational > temporal > logical > coincidental

High cohesion — Low cohesion

Types of Coupling

- Coupling is the type and amount of interaction between modules
- Coupling among modules
  - module A and B access to the same global variable
  - module A calls module B with some arguments
- Arbitrary modularization will result in tight coupling
  - Loosely coupled modules are good, tightly coupled modules are bad
- If you use only one module, you get no coupling. Good idea?
  - No! You did not reduce the complexity of the system. You did not modularize.

Types of Bad Coupling

- Common (or Global) coupling
  - Access to a common global data by multiple modules
  - Class variables are also a limited form of common coupling, use them with caution

- This is a bad type of coupling: The interactions among the modules are through global data so it is very difficult to understand their interfaces and interactions. It is hard to debug, and maintain such code.

Types of Bad Coupling


```
int number_of_students
float student_grades[];

procedure find_maximum_grade(student_grades)
{
// traverse the array student_grades from 0 to number_of_students
// to find the maximum grade
}


procedure find_minimum_grade(student_grades)
{
// traverse the array student_grades from 0 to number_of_students
// to find the minimum grade
}
```

Types of Bad Coupling

- Control coupling
  - If one module passes an element of control to another module
  - For example a flag passed by one module to another controls the logic of the other module
- This type of code is hard to understand
  - It is hard to understand the interfaces among the modules, you need to look at the functionality to understand the interfaces

Types of Bad Coupling


```
call operations (d1, d2, opcode);

procedure operations (data1, data2, operation)
{
  switch (operation) {
    case ...: // execute operation 1 on data1
    case ...: // execute operation 2 on data2
  }
}
```

Good coupling

- Data coupling
  - The interaction between the modules is through arguments passed between modules
  - The arguments passed are homogenous data items

- Data coupling is the best type of coupling

- In the data coupling you should try to pass only the parts of data that is going to be used by the receiving module
  - do not pass redundant parts

Modularization

- Complexity
  - A design with complex modules is worse than a design with simpler modules
  - Remember the initial motivation in modularization is to reduce the complexity
  - If your modules are complex this means that you did not modularize enough
  - Modularization means using divide-and-conquer approach to reduce complexity

Modularization

- Now we will discuss and compare two modularization strategies

- These modularization strategies are both intended to generate modules with high cohesion and low coupling

  – Modularization technique 1: Functional decomposition

  – Modularization technique 2: Parnas's modularization technique
    "On the Criteria to be Used in Decomposing Systems into Modules", Parnas 1972

Functional Decomposition

- Functional decomposition
  - Divide and conquer approach
  - Use stepwise refinement
    1. Clearly state the intended function
    2. Divide the function to a set of sub-functions and re-express the intended function as an equivalent structure of properly connected sub-functions, each solving part of the problem
    3. Divide each sub-function far enough until the complexity of each sub-function is manageable

- How do you divide a function to a set of sub-functions? What is the criteria? This approach does not specify the criteria for decomposition
  - Based on how you decompose the system the modules will show different types of cohesion and coupling

Functional Decomposition

- One way of achieving functional decomposition: Make each step in the computation a separate module
  - Draw a flowchart showing the steps of the computation and convert steps of the computation to modules
  - Shortcoming: Does not specify the granularity of each step
- Another way of achieving functional decomposition is to look at the data flows in the system
  - Represent the system as a set of processes that modify data. Each process takes some data as input and produces some data as output.
  - Each process becomes a module
- Shortcoming: Both of these approaches produce a network of modules, not a hierarchy

What about Data Structures?

- Fred Brooks: *"Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious."*
- Eric Stevens Raymond: *"Smart data structures and dumb code works a lot better than the other way around."*

- Functional decomposition focuses on operations performed on data
- According to Brooks and Raymond data structures should come first
- Parnas's modularization approach pays attention to data

Example:
KWIC Index Production System

- The KWIC (Key Word In Context) index system
  - Accepts an ordered set of lines
  - Each line is an ordered set of words, and each word is an ordered set of characters.
  - Any line may be “circularly shifted” by repeatedly removing the first word and appending it at the end of the line.
  - The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

KWIC: Input and Output

Input:
to understand better
need an example

Output:
an example need
better to understand
example need an
need an example
to understand better
understand better to

Modularization 1

- Use functional decomposition
- Generate five modules based on the functionality

1. **Input:** Read the data lines from the input and store them internally in an array that will be accessed by other modules

2. **Circular Shifter:** Called after the input module finishes its work. Prepares an array of all circular shifts: Each array item is a pair (original line number, location of the first character of the circular shift)

3. **Alphabetizing:** Using the arrays produced by the first two modules this module produces an array in the same format produced by module 2 but the array is ordered based on the alphabetical order

Modularization 1

- Remaining modules

4. **Output:** Using the arrays produced by modules 1 and 3 it produces the output listing of all the circular shifts in alphabetical order

5. **Master Control:** Controls the sequencing among the other four modules, can also handle error messages, space allocation etc.

Modularization 1

Module 5
Master Control ← Decides the control flow
handles error messages etc.

Reads the input and creates an array of strings	Generates an array listing all the circular shifts	Sorts the circular shift array	Generates the output
↓	↓	↓	↓
Module 1 Input	Module 2 Circular-Shifter	Module 3 Alphabetizer	Module 4 Output
1 "to understand better"	(1, 1)	(2,6)	an example need
2 "need an example"	(1, 4)	(1,15)	better to understand
	(1,15)	(2,9)	example need an
line number →	(2,1)	(2,1)	need an example
	(2,6)	(1,1)	to understand better
character location in line (start from that character and wrap around to get the circular shift) ↗	(2,9)	(1,15)	understand better to

Modularization 2

- **Input:** Reads the input and calls the Line Storage module to have them stored internally

- **Line Storage:** This module consists of a number of procedures such as

  CHAR(*r*,*w*,*c*): returns the *c*th character in the *w*th word of the *r*th line

  WORDS(*r*): returns as value the number of words in line *r*

  …

  All the above procedures have certain restrictions in the way they can be called and if these restrictions are violated they raise an exception that should be handled by the caller

Modularization 2

- **Circular Shifter:** Provides essentially the same functionality as the Line Storage module. It creates the impression that not the original lines but all the circular shifts of the original lines are stored

  CS_CHAR(*l*,*w*,*c*): returns the *c*th character in the *w*th word of the *l*th circular-shift

  It is specified that

  (1) if i < j then the shifts of line i precede the shifts of line j

  (2) for each line the first shift is the original line, the second shift is obtained by making a one-word rotation to the first shift, etc.

  A function CSSETUP is provided which must be called before other functions have their specified values

Modularization 2

- **Alphabetizer:** Provides two functions

  ALPH must be called before the other function has a meaningful value

  ITH(*i*): Gives the index of the circular-shift that comes *i*th in the alphabetical ordering

- **Output:** Prints the circular-shifts in alphabetical order

- **Master Control:** Similar to the previous modularization, controls the sequencing among the other four modules

Comparison

- Both approaches would work
- Actually, the generated binary code for both approaches might be identical
- The differences are in the way they are divided to different modules and the interfaces between modules
- Binary representation is for running the program on a machine
  - However other representations (design specification, source code) is for changing, documenting, and understanding. Two systems will not be same in terms of these other representations

Changeability

There are a number of design decisions that could change:

1. Input format
2. The decision to store all lines in main memory
    - It may be necessary to use disk storage for some applications
3. Storage format for the string can change
4. Using an index array for circular shifts.
    - It may be better to store them as strings for some applications.
5. Alphabetize the list at once rather than searching for an item when needed, or partially alphabetize

Changeability

Now, let's look at how different modularizations perform under these changes:

- The 1st change (changing the input format) is confined to the **Input** module in both modularizations
- The 2nd and 3rd changes (not storing lines in memory and changing the storage format for strings) will require changing every module in the first modularization. Every module accesses the storage format of the lines and strings. The situation is completely different for the second modularization, only the **Line Storage** module has to change:
  - The knowledge of exact way the lines are stored is entirely hidden from all but the **Line Storage** module

Changeability

- The 4th change (not using an index array for circular shifts) is confined to the **Circular-shifter** module in the second decomposition but in the first decomposition in addition to **Circular-shifter**, **Alphabetizer** and **Output** modules will have to change too

- The 5th change (alphabetizing incrementally or partially) will also be difficult for the first decomposition. Since the **Output** module expects the index to have been completed before it began, this change will not be confined to the **Alphabetizer** module. In the second decomposition the user of the **Alphabetizer** module (i.e., the output module) does not know exactly when the alphabetizing is done so the modification will be confined to the **Alphabetizer** module.

Independent Development

- In the first modularization the interfaces between the modules are complex involving arrays for strings and index arrays
  - The design of these data structures are important for the efficiency of the system. The design of the data structures will involve all development groups working on different modules

- In the second decomposition interfaces are abstract, they involve function names and types and number of parameters
  - These decisions are much easier. Hence, independent development of the modules can begin much earlier

Comprehensibility

- To understand the code of the different modules in the first decomposition one has to understand the storage formats.
- In the second decomposition this is only necessary for understanding the line storage module.
  - The rest of the modules can be understood without understanding how data is stored.

First Modularization

- Functional decomposition
- Makes each step in the computation a separate module
  - Draw a flowchart showing the steps of the computation and convert steps of the computation to modules

Second Modularization

- Every module in the second decomposition is characterized by its knowledge of a design decision which it hides from others.
  - Its interface or definition is chosen to reveal as little as possible about its inner workings
  - This principle is called **Information Hiding**
- Modules do not correspond to steps in the computation
- A data structure, its internal representation, and accessing and modifying procedures for that data structure are part of a single module

What about Efficiency?

- There will be too many procedure calls in the second approach which may degrade the performance
  - Use inline expansion, insert the code for a procedure at the site of the procedure call to save the procedure call overhead
  - This is a common compiler optimization

Parnas ~ Object Oriented Design

- In his paper on modularization, even without an object-oriented programming language, Parnas advocates principles of object-oriented design and programming
  - Information hiding
  - Encapsulation: line storage module encapsulates the data and the functionality
  - Abstraction: Circular shift module is a specialization of the line storage module
  - Inheritance: Circular shift module can inherit some functionality from the line storage module
  - Polymorphism: In the “Hierarchical Structure” section Parnas talks about having a parameterized version of the system where either circular shift or the original line storage module is used
- All of these features are supported by modern object-oriented languages such as C++ and Java

Modularization Summary

- The goals of modularization are to ***reduce the complexity*** of the software, and to ***improve maintainability, reusability and productivity***.
- A module is a responsibility assignment rather than a subprogram.
- Good modularization: ***highly cohesive modules*** and ***low coupling between modules***
- One modularization approach:
  - ***Functional decomposition***: Draw a flowchart showing the steps of the computation and convert steps of the computation to modules.
- Better modularization approach:
  - ***Information hiding***: Isolate the changeable parts, make each changeable part a secret for a module. Module interface should not reveal module's secrets.