UECS2344 Software Design: Lecture 3

Modularity, Abstraction, and Responsibility-Driven Development

Modularity

Modularity is about dividing an application into separately named parts called **modules**.
Each module is integrated into the application and contributes to its overall work.

Modularity helps manage the complexity of an application and makes it easier to understand, develop, and test the application.

Modularity in Structured Approach

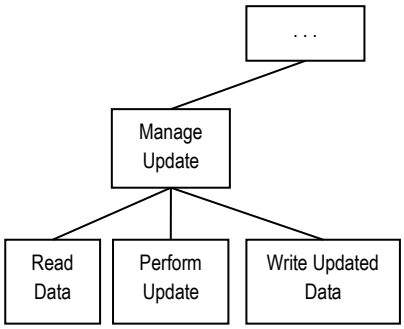
In the structured programming approach, a function can be considered a module.

When we develop a large application, we do not write a single large main function to do all the work, i.e. we do not write a *monolithic* program - a program composed of a single module.

Instead we write various functions to carry out different parts of the work. The main function then calls these functions in the correct order to accomplish the overall work.

Modularity is achieved through **Modular / Functional / Hierarchical Decomposition**.
A modular decomposition design is shown in a **Structure Chart**.

Example:




```
void manageUpdate()
{
    readData(...);
    performUpdate(...);
    writeUpdatedData(...);
}
```

Criteria for Modular Decomposition

Cohesion – the degree of relatedness of *elements within a module*

Coupling – the degree of relatedness *between modules*

For good software design:

- **aim for High Cohesion:** a module should have a single, well-defined purpose and all elements within a module should contribute to that purpose and to that purpose only; therefore aim for high degree of relatedness of all the elements within the module.

- **aim for Low Coupling:** a module should be as independent as possible from other modules or, in other words, it should be loosely coupled to other modules; it should not have

unnecessary dependencies on other modules to accomplish its purpose; therefore aim for low degree of relatedness to other modules.

More on Cohesion and Coupling
(adapted from https://courses.p2pu.org/en/courses/1099/content/2343/)

Cohesion

When designing a function, we want each line in the function to directly relate to the purpose of that function. If we have pieces of code within the function which do not relate to the purpose of the function, we should pull out the unrelated code to form to a new function.

Note: The purpose of the function should be reflected in the name of the function. Naming functions correctly is very important to improve readability of an application.

High cohesion improves development and testing. It also improves readability (and therefore, maintainability).

Coupling

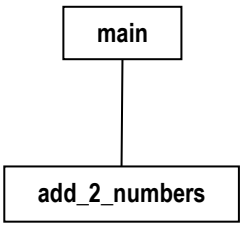
A function should have minimal dependence on other functions and be highly independent, or in other words, they should have low coupling to other functions.

Low coupling improves maintainability because a change in one function will have a minimal effect on other functions. It also promotes reusability because independent functions are more easily reused in other applications.

Example of Improving Program Design

Version 1 (bad):

```
int main(void) {

      add_2_numbers();

      return 0;
}

void add_2_numbers() {
      int num1, num2;

      cout << "Enter a number: ";
      cin >> num1;
      cout << "Enter a number: ";
      cin >> num2;

      sum  = num_1 + num_2;

      cout << "The sum is " << sum;
}
```

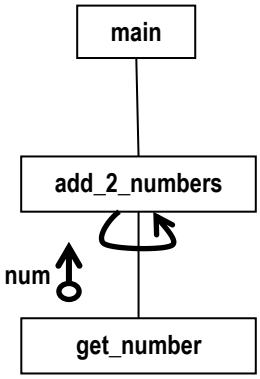


Look at function add2numbers().

Notice the function is **not cohesive** – it is not doing just one thing i.e. adding 2 numbers as suggested by its name – instead it is adding 2 numbers and also getting input and producing output.
Suppose we rewrite the program and create a new function to get the input as follows:

Version 2 (still bad):

```
int main(void) {

	add_2_numbers();

	return 0;
}

int get_number() {
	cout << "Enter a number: ";
	cin >> num;
	return num;
}

void add_2_numbers() {
	num_1 = get_number();
	num_2 = get_number();

	sum  = num_1 + num_2;

	cout << "The sum is " << sum;
}
```



Now function add_2_numbers() is **coupled** to (or **dependent** on) the function get_number(), i.e. add_2_numbers() cannot work without get_number(). Also, if we want to use function add_2_numbers in another program, we have to copy both add_2_numbers and get_number() into the new program.

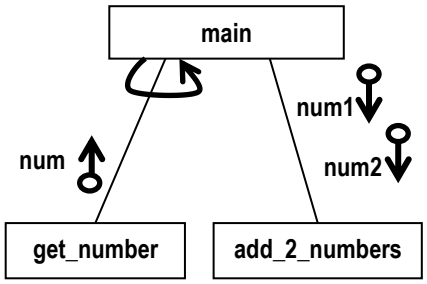
Note: the dependency is one-way – add_2_numbers() is dependent on get_number() but get_number() is not dependent on add_2_numbers().

Suppose we rewrite the program as follows:

Version 3 (better):

```
int main(void) {

	int num1, num2;

	num_1 = get_number();
	num_2 = get_number();

	add_2_numbers(num1, num2);

	return 0;
}
```

```
int get_number() {
      cout << "Enter a number: ";
      cin >> num;
      return num;
}

void add_2_numbers(int n1, int n2) {

      sum  = n1 + n2

      cout << "The sum is " << sum;
}
```



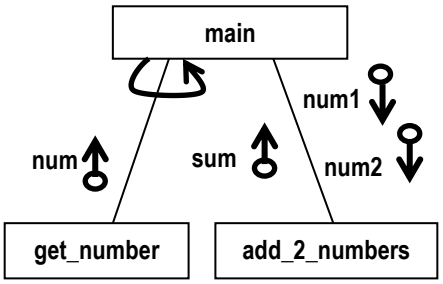
Now function add_2_numbers() is better – it doesn't depend on get_number() anymore to get the numbers to add. There is **no coupling** or **dependency** on get_number().

The purpose of function add_2_numbers should simply be to add 2 numbers which are passed to it as parameters. However, in addition to adding 2 numbers, it is also producing output. So function add_2_numbers() is still **not cohesive**. Not all the statements in the function are related to its purpose of adding 2 numbers only.

Function add_2_numbers should return the result of the addition and leave it to function main() to decide how it wants to use this result. We rewrite the program as follows:

Version 4 (good):

```
int main(void) {

      int num1, num2, sum;

      num_1 = get_number();
      num_2 = get_number();

      sum = add_2_numbers(num1, num2);

      cout << "The sum is " << sum;

      return 0;
}

int get_number() {
      cout << "Enter a number: ";
      cin >> num;
      return num;
}

void add_2_numbers(int n1, int n2) {
      return n1 + n2;
}
```



Now function add_2_numbers() is only adding two numbers. There is no other code in the function that is not directly related to the purpose of addition and so, we can say that add_2_numbers is **highly cohesive**.

Exercise 1

Consider the following payroll processing program.

It is a monolithic program, consisting of only one function.

Perform modular decomposition (focus on the report writing operations).

Produce a Structure Chart to represent the new design of the program.

```
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>

using namespace std;


struct EmployeePay
{
	int empId;
	double grossPay;
	double deductions;
	double netPay;
};

int main(void)
{
	ofstream report;
	string filename;

	double totalGrossPay;
	double totalDeductions;
	double totalNetPay;

	char choice;

	EmployeePay empPay;


	totalGrossPay = 0.0;
	totalDeductions = 0.0;
	totalNetPay = 0.0;

	cout << "Please enter report file name: ";
	cin >> filename;
	report.open(filename);

	report << "==============Payroll Report================" << endl;
	report << "    Emp-ID   Gross Pay  Deductions Net Pay" << endl;
	report << "==========================================" << endl;
```

```
	choice = 'y';

	do
	{
		cout << "Enter payroll number for employee: ";
		cin >> empPay.empId;
		cout << "Enter gross pay: ";
		cin >> empPay.grossPay;
		cout << "Enter deductions: ";
		cin >> empPay.deductions;

		empPay.netPay = empPay.grossPay - empPay.deductions;

		totalGrossPay += empPay.grossPay;
		totalDeductions += empPay.deductions;
		totalNetPay += empPay.netPay;

		report << setw(10) << empPay.empId
			<< setw(10) << empPay.grossPay
			<< setw(10) << empPay.deductions
			<< setw(10) << empPay.netPay << endl;
		report << "-----------------------------------------"
			<< endl;

		cout << "Any more employees (y/n)? ";
		cin >> choice;

	} while (choice == 'y');

	report << endl;
	report << "Total Gross Pay:  " << setw(6)
		<< totalGrossPay << endl;
	report << "Total Deductions: " << setw(6)
		<< totalDeductions
		<< endl;
	report << "Total Net Pay:    " << setw(6)
		<< totalNetPay << endl;

	report.close();

	cout << "Report created successfully" << endl;

}
```

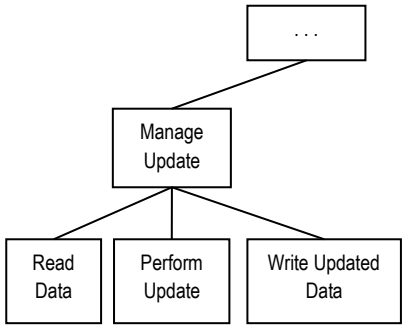
Abstraction

An **Abstraction** is a representation of something with only the relevant details included; other details that are irrelevant are ignored.

Abstraction in Structured Approach

A function-call is a **Procedural Abstraction** because it represents only certain details of a function: the name, the parameters (if any), and the return value (if any). These are the relevant details of the function in a function-call. The implementation details of the function, i.e. how the function works or how it is implemented) is not relevant when we call the function.

Example:




```
void manageUpdate()
{
    readData(...);
    performUpdate(...);
    writeUpdatedData(...);
}
```

In function manageUpdate(), we call the three functions readData(), performUpdate() and writeUpdatedData(). We don't need to know how these functions are implemented when calling them; the only details we need are the function names and details about the parameters and return values (if any).

Modularity and Abstraction in Object-Oriented Approach

In the object-oriented approach, a class can be considered a **module**. An object-oriented application is composed on one or more classes.

A class is also an **abstraction** because it represents something in the real-world, for example, a Student class represents a student in the real-world.

Responsibility-Driven Design – developed by Rebecca Wirfs-Brock and Brian Wilkerson

(adapted from http://www.wirfs-brock.com/Design.html)

Responsibility-Driven Design is a way of thinking about class as responsibilities.

In responsibility-driven design, each class is responsible for a specific portion of the work of the application, i.e. we assign specific responsibilities to each class.

Each class should be responsible for (knows about and does) only a few related things – this makes the class highly cohesive.

Information and behaviour related to a responsibility should be placed within the same class – this is the object-oriented concept of encapsulation.

"Understanding responsibilities is key to good object-oriented design" – Martin Fowler

Exercise 2

For the payroll processing program above, develop an object-oriented version.

- **You are to identify the classes you need and assign responsibilities to them.**
- **Produce a Class Diagram to represent the design.**