Programming II

Modularity

Lecture Outline

- Evolution and history of programming languages
- Modularity
- Example

History of Programming

Programming Paradigms

- How and why languages develop?
- How the OOP differs from previous paradigms?
- Aspects of software quality.

Large Projects

- Programming of large software projects qualitatively differs from creating of small programs.

- The reason for thinking about the principles of programming is the increasing complexity of computer programs.

Paradigms

- Imperative programming
- Declarative programming
  - Functional programming
  - Logic programming
- Modular programming
- Object oriented programming

Imperative Programming

- The sequence of steps (statements) by which we change the state of program variables.
- Structured programming
  - Sequence, iteration (loops), branching (selection), jumps, recursion and abstraction.
- Procedural programming
  - Derived from structured programming, based upon the concept of the procedure (subroutine, function) call.
- As we know from the most commonly used languages.

Modular Programming

- Top-down design of code.
- Division of the program into independent, interchangeable modules that provide partial functionality.
- The module contains everything necessary to ensure the functionality (data and algorithms).

Object Oriented Programming

Factors of Software Quality

- Internal and external factors.
- Internal factors are hidden from the user. (HOW)
- External factors describe the external behavior. (WHAT)

External Factors

- Correctness
- Robustness
- Reusability, extendibility, usability, compatibility,...

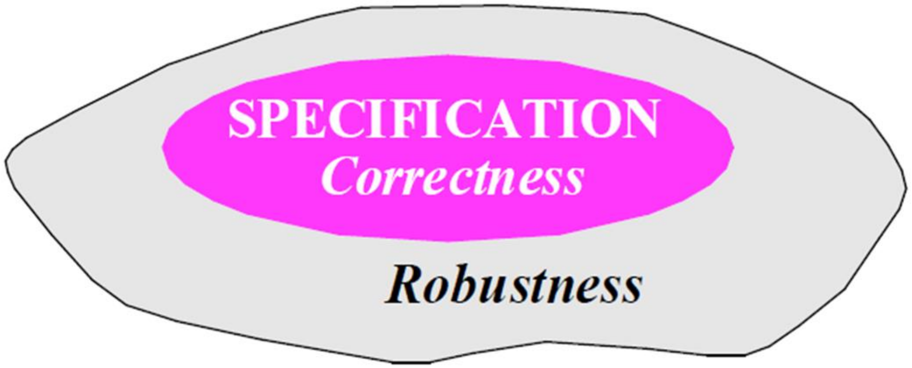
Key Factors

Definition: correctness

Correctness is the ability of software products to perform their exact tasks, as defined by their specification.

Definition: robustness

Robustness is the ability of software systems to react appropriately to abnormal conditions.



Other Factors

Definition: extendibility

Extendibility is the ease of adapting software products to changes of specification.

Definition: reusability

Reusability is the ability of software elements to serve for the construction of many different applications.

Definition: compatibility

Compatibility is the ease of combining software elements with others.

Definition: functionality

Functionality is the extent of possibilities provided by a system.

Modularity

Modularity of Program

- Understandability
- Independence
- Composability
- Encapsulation
- Explicit interface
- Syntactic support

Understandability

- The module should provide a clearly defined and simple task, or a few clearly defined tasks.

Independence

- Each module must be relatively independent, and should have the minimum possible number of links to other modules.
- It would not be appropriate for all program modules to be interconnected and interdependent.
- The modules cannot be individually tested, understand or be transferred to another project.

Composability

- The modules need to be combined with each other. It must be possible to take the module and use it in another context, or even in another project.

Encapsulation

- Modules must have some privacy; it is desirable that any information that is not needed for clients of modules should be hidden inside the module.
- In practice, it shows that most of the functionality of the module are hidden, and only a small part is visible externally.
- Hidden parts of a module are often called module implementation and public parts are called interface of the module.

Explicit Interface

- It must be universally understood what assumptions module needs to perform its tasks.

Syntactic Support

- The modules of a computer program must be clearly defined as syntactic units of the program.

- From the source code of the programming language must be quite clear where a single module begins and ends.

Five Criteria and Rules

- Decomposability.
- Composability.
- Understandability.
- Continuity.
- Protection.

- Direct Mapping.
- Few Interfaces.
- Small interfaces (weak coupling).
- Explicit Interfaces.
- Information Hiding.

Criteria

A software construction method satisfies Modular Decomposability if it helps in the task of decomposing a software problem into a small number of less complex subproblems, connected by a simple structure, and independent enough to allow further work to proceed separately on each of them

A method satisfies Modular Composability if it favors the production of software elements which may then be freely combined with each other to produce new systems, possibly in an environment quite different from the one in which they were initially developed.

A method favors Modular Understandability if it helps produce software in which a human reader can understand each module without having to know the others, or, at worst, by having to examine only a few of the others.

Criteria

A method satisfies Modular Continuity if, in the software architectures that it yields, a small change in a problem specification will trigger a change of just one module, or a small number of modules.

A method satisfies Modular Protection if it yields architectures in which the effect of an abnormal condition occurring at run time in a module will remain confined to that module, or at worst will only propagate to a few neighboring modules.

Rules

The modular structure devised in the process of building a software system should remain compatible with any modular structure devised in the process of modeling the problem domain.

Every module should communicate with as few others as possible.

If two modules communicate, they should exchange as little information as possible

Whenever two modules *A* and *B* communicate, this must be obvious from the text of *A* or *B* or both.

The designer of every module must select a subset of the module's properties as the official information about the module, to be made available to authors of client modules.

Example

Class x Object

- Class as a static description.
- Object as a run-time representation:
  - state (data)
  - Behavior (algorithms)

Terminology

- Class
- Member function, method
- Member variable
- Object, instance of a class
- Constructor, destructor

Class Declaration

```
#include <iostream>
using namespace std;

class KeyValue {
private:
    int key;
    float value;
    KeyValue *next;

public:
    KeyValue(int k, float v);
    ~KeyValue();
    float GetValue();
    void CreateNext(int k, float v);
    KeyValue* GetNext();
};
```

Class Implementation (definition)

```
KeyValue::KeyValue(int k, float v) {
    this->key = k;
    this->value = v;
    this->next = NULL;
}

KeyValue::~KeyValue() {
    if(this->next != NULL){
        delete this->next;
    }
}

float KeyValue::GetValue() {
    return this->value;
}

void KeyValue::CreateNext(int k, float v) {
    this->next = new KeyValue(k, v);
}

KeyValue* KeyValue::GetNext() {
    return this->next;
}
```

Using the Class

```
int main() {
    KeyValue *c1 = new KeyValue(1, 1.5);
    cout << c1->GetValue() << endl;

    c1->CreateNext(2, 2.5);
    cout << c1->GetNext()->GetValue() << endl;
    delete c1;

    getchar();
    return 0;
}
```

Questions

- What is the main motive for developing programming paradigms from the imperative to the object oriented?
- What is imperative programming?
- What is modular programming?
- What are the main factors of software quality?
- What understandability of module?
- What is the independence of module?
- What is composability of modules?
- What is the encapsulation of module?
- What is the explicit interface of a module?
- What is the syntactic support for modularity?
- What are the five criteria for good modularity?
- What is meant by the five rules of good modularity?

Sources

- *History of programming languages*. Wikipedia. https://en.wikipedia.org/wiki/History_of_programming_languages#Establishing_fundamental_paradigms

- Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall 1997. [3-16, 39-52]