Programmimg Languages:
Encapsulation

Onur Tolga Şehitoğlu

Computer Engineering

20 March 2008
1. Encapsulation

2. Packages

3. Hiding

4. Abstract Data Types

5. Class and Object
   - Object
   - Class
Encapsulation

Managing the complexity → Re-usable code and abstraction.
Example:

<table>
<thead>
<tr>
<th>Lines</th>
<th>Abstraction Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>no abstraction is essential, all in main()</td>
</tr>
<tr>
<td>500</td>
<td>function/procedure abstraction sufficient</td>
</tr>
<tr>
<td>5,000</td>
<td>function groups forming modules, modules are combined to form the application</td>
</tr>
<tr>
<td>500,000</td>
<td>heavy abstraction and modularization, all parts designed for reuse (libraries, components etc)</td>
</tr>
</tbody>
</table>
Modularization and Encapsulation

- Building an independent and self complete set of function and variable declarations (**Packaging**)
- Restricting access to this set only via a set of interface function and variables. (**Hiding and Encapsulation**)

Diagram:
- Interface
- Detail (functions, variables, algorithm)
- Other application
Advantages of Encapsulation

- High volume details reduced to interface definitions (Ease of development/maintenance)
- Many different applications use the same module via the same interface (Code re-usability)
- Lego like development of code with building blocks (Ease of development/maintenance)
- Even details change, applications do not change (as long as interface is kept same) (Ease of development/maintenance)
- Module can be used in following projects (Code re-usability)
- A group of declarations put into a single body.
- C has indirect way of packaging per source file.
- C++

```cpp
namespace Trig {
    const double pi = 3.14159265358979;
    double sin(double x) { ... }
    double cos(double x) { ... }
    double tan(double x) { ... }
    double atan(double x) { ... }
    ...
}
```

- Trig::sin(Trig::pi/2+x)+Trig::cos(x)
- C++: (::) Scope operator.
- Identifier overlap is avoided. List::insert(...) ve Tree::insert(...) no name collisions.
A group of functions and variables hidden inside. The others are interface. Abstraction inside of a package:

```haskell
module Trig (sin, pi, cos) where
    taylorseries x = ...
    sin x = ...
    pi = 3.14159265358979
    randomseed = ...
    cos x = ...
    errorcorrect x = ...
```
Abstract data types

- Internals of the datatype is hidden and only interface functions provide the access.

- Example: rational numbers: 3/4, 2/5, 19/3

  data Rational = Rat (Integer,Integer)
  x = Rat (3,4)
  add (Rat(a,b)) (Rat(c,d)) = Rat (a*d+b*c,b*d)

  1 Invalid value? Rat (3,0)
  2 Multiple representations of the same value?
    Rat (2,4) = Rat (1,2) = Rat (3,6)

- Solution: avoid arbitrary values by the user.
Main purpose of abstraction data types is to use them transparently (as if they were built-in) without losing data integrity.

```haskell
module Rational (Rational, rat, add, subtract, multiply, divide) where
  data Rational = Rat (Integer, Integer)
  rat (x,y) = simplify (Rat(x,y))
  add (Rat(a,b)) (Rat(c,d)) = rat (a*d+b*c,b*d)
  subtract (Rat(a,b)) (Rat(c,d)) = rat (a*d-b*c,b*d)
  multiply (Rat(a,b)) (Rat(c,d)) = rat (a*c,b*d)
  divide (Rat(a,b)) (Rat(c,d)) = rat (a*d,b*c)
  gcd x y = if (x==0) then y
              else if (y==0) then x
              else if (x<y) then gcd x (y-x)
              else gcd y (x-y)
  simplify (Rat(x,y)) = if y==0 then Rat(div x y,1)
                        else let a=gcd x y
                             in Rat(div x a, div y a)
```

Initial value? We need constructor function/values. (remember we don’t have the data definition)
rat (x,y) instead of Rat (x,y)
Object

- Packages containing hidden variables and access is restricted to interface functions.
- Variables with state
- Data integrity and abstraction provided by the interface functions.
- Entities in software can be modelled in terms of functions (server, customer record, document content, etc). Object oriented design.

Example (invalid syntax! imaginary C++)

```cpp
namespace Counter {
private: int counter=0;
public:  int get() { return counter; }
public:  void increment() { counter++; }
};
Counter::get()    Counter::increment()
```
Class

- The set of same typed objects form a class
- An object is an instance of the class that it belongs to (a counter type instead of a single counter)
- Classes have similar purposes to abstract data types
- Some languages allows both objects and classes
- C++ class declaration (valid syntax):

```c++
class Counter {
private:   int counter;
public:    Counter() { counter=0; }
            int get() { return counter; }
            void increment() { counter++; }
} men, vehicles;
men.increment(); vehicles.increment();
men.get(); vehicles.get();
```
Abstract data type

- **interface** (constructor, functions)
- **detail** *(data type definition, auxiliary functions)*

Object

- **interface** (constructor, functions)
- **detail** *(variables, auxiliary functions)*

**Purpose**

- preserving data integrity,
- abstraction,
- re-usable codes.