Encapsulation in C++

Copyright © Jim Fawcett
Spring 2017
Encapsulation

• Encapsulation Requires that functions, modules and classes:
  • Have clearly defined external interfaces
  • Hide implementation details

• Encapsulation is all about coupling – coupling happens through interfaces.

• Exactly what is an interface?
Function Interfaces

• Functions have four ways to interact with the external world:
  • Parameter list – local coupling:
    • Value parameters and const references are input only
      • Functions with these parameters don’t have parametric side-affects.
    • Non-const reference parameters and pointers are both input and output
      • We say that a function with non-const reference parameters has side-affects.
  • Returned items – local coupling:
    • Return by value has no after-affects inside the function.
    • Return by reference can change the state of an object to which the function belongs.
  • Global data – remote coupling:
    • Functions that use global data create remote and untraceable side effects.
  • Static local data – temporal coupling:
    • An invocation affects later invocations.

• The strongest encapsulation is with only pass-by-value or constant reference in and out.

• However, the indirection allowed by non-constant references is just too powerful. We can’t live without it.

• We can, and should, live without global data and we should minimize the use of static local data.
C++ References versus Pointers

• Prefer returning C++ references over pointers.
  • References provide access to objects. You only get to use the object’s interface with a reference.
  • Pointers provide access to memory. Their proper use demands that the client understand the design of the function, and perhaps the class:
    • Does the pointer point to an object or an array of objects?
    • Does it point to the heap or some other persistent object?
    • Whose responsibility is destruction?

• This is not a manifesto to eliminate all pointer use!
  • Pointers do a great job of capturing relationships:
    • Arrays, Graphs, trees, lists, repositories
    • Their big brothers, iterators, are an essential part of the STL
  • It is appropriate to return them from creational functions that act like sophisticated new calls.
Classes

• All the same issues hold for classes, via the member functions they provide.

• Classes also support three accessibility levels: public, protected, and private.
  • Public members define the client interface and should be immutable – changing the interface breaks client designs.
  • Protected members define an interface for derived classes only. They also should be immutable once some of the derived classes are no longer under your control.
  • Private members consist of all those helper functions that manage the class’s internal state.

• For proper class encapsulation:
  • Don’t use public or global data
  • Encapsulate member functions well, as discussed in earlier slides.
  • Decide carefully which functions will be public, protected, and private.
  • Never make member data public.
Role of Constructors and Destructor

• Constructors and destructors
  • Objects are created from a class pattern only by calling a constructor.
  • Whenever the thread of execution leaves a scope, all objects created in that scope are destroyed. Part of that destruction is the execution of the objects’ destructor and the destructors of all its parts.
  • A constructor’s job is to allocate whatever resources the class needs on startup.
  • The destructor returns any allocated resources.
  • These actions mean that the class encapsulates its own resource management. That is a very big deal!
Class Relationships

- Composition is the best encapsulated relationship:
  - Only member functions of the class have access to the interface of private data member objects.
  - Only member functions of the class and its derived classes have access to the interface of a protected data member object.

- Inheritance gives derived classes access to the protected data members of the base class.
  - Behavior of a correct base class can be made incorrect by incorrect derived class objects.

- Using relationships may badly break encapsulation.
  - If you pass an object of a class to a member function of an object of the same class, the receiving object has access to all the used object’s private and protected state.
  - Friend relationships extend this behavior to objects of other classes.
Class Relationships

Object Relationships

client

Aggregation

friend

Composition

Is-A

uses

uses

aggregation

(ab)uses?

client

friend

b ∈ B
c ∈ C
d ∈ D

u ∈ U

B

C

D

U

Composition

Is-A

uses
Bad Designs

• What makes a design bad? Robert Martin suggests[1]:
  
  • **Rigidity**  
    It is hard to change because every change affects too many other parts of the system.
  
  • **Fragility**  
    When you make a change, unexpected parts of the system break.
  
  • **Immobility**  
    It is hard to reuse a part of the existing software in another application because it cannot be disentangled from the current application.

• Many of these “Bad Design” issues stem very directly from poor encapsulation.
C++ Class Encapsulation Problem

• The C++ object model has the source text of a user of an object responsible for carrying the object’s type information by including a header file.
  • If class A refers to class B, then A must include B’s header.
  • This is the way a C++ compiler does static type checking.
  • Remember, in contrast, that C# carries its type information in the assembly metadata of the object itself.

• A direct consequence of this fact is that, unless we do something special to prevent it, the client is welded to specific versions of the objects it uses.
  • If the object changes anything in its class declaration, adding a new member datum, for example, then the client has to acquire its new header and recompile.
Programming to Interfaces

• The something special we do, mentioned on the previous slide, is to:
  • Program to an Interface not an Implementation
  • Use an Object Factory to create the object.

• Here, interface means more than the public interface of a class.
  • An interface is an abstract base class, often a struct actually, containing at least one pure virtual function.
  • It usually has no constructors and no data.
  • It has a destructor, only to qualify it as virtual.
  • It often has a static factory function so the client does not have to directly instantiate an object of the class it represents.

• This interface shields its clients from all of the implementation details of the class it represents.
Client Using IElement Interface to Build Tagged Elements

**IElement**

Public:
- static IElement* CreateElement(const std::string &tag);
- virtual void addAttribute(const std::string &name, const std::string &value) = 0;
- virtual void addBody(const std::string &body) = 0;
- virtual std::string makeElement() = 0;

Private:
- std::string _tag;
- std::map<std::string, std::string> _attribs;
- std::string _body;

**Element**

Public:
- Element(const std::string &tag);
- virtual void addAttribute(const std::string &name, const std::string &value);
- virtual void addBody(const std::string &body);
- virtual std::string makeElement();

Client does not include the Element.h header file, only the IElement header, so it has no compilation dependencies on Element class.

Static creational function is declared in the interface, but implemented in Element.cpp file, where it sees Element.h class declaration.

Element class has a constructor, used by CreateElement(...) to create and return a pointer to a new Element object.

Client Using IElement Interface to Build Tagged Elements
Fini

• Designing to Interfaces and using factory functions completely eliminates the coupling of client to the implementation of its serving objects.
End of Presentation