Chapter 10 – Design Notes

Jim Fawcett

Table of Contents

What is Design?

- Design is the process of deciding how to satisfy a program's requirements.

- Design has four essential elements:
  - ***client focus***
    concerned with how the user will interact with the program.
  - ***organizing principles***
    the main design ideas on which a program's implementation is based.
  - ***Structure***
    the physical way that code is formed for implementation. The design issues here are:
    - the parts into which the program is divided.
    - communication required between parts.
    - ownership of system resources and other parts.
    - visibility of one part by another.
  - ***Performance***
    determined by:
    - algorithms used in implementing requirements
    - how often the memory manager is called
    - how arguments of functions are passed
    - which objects are static
    - which objects are made local
    - the size of objects and the frequency of their construction and copying

Client Focus

- A designer's first responsibility is to the user of the program and that should be his first and primary concern during the initial stages of design.
- Important questions are:
  - how can I minimize the amount of information the user has to supply the program?
  - how can I make the program flexible enough to allow the user to fully satisfy his needs?
  - how can I inform the user about inputs the program needs?
  - how can I inform the user about progress of the computation?
  - how can I help the user understand what the program is doing for him?
  - how can I help the user interpret program's results?
- Note that the user may be a human being or it may be another software component. The same questions are appropriate for either case.

Principles of Awful Interface Design

- Design: wait for user input without prompting
- Principle: user will figure it out eventually

- Design: prompt for input without indicating how to supply it
- Principle: everyone needs a little frustration

- Design: change state of the system without warning
- Principle: users love surprises

- Design: when an error occurs tell the user an error occurred
- Principle: don't bother the user with details, especially if useful

- Design: provide data instead of information. Dump raw numbers on the output without labels or interpretation
- Principle: do what is easy for the designer, not the user

Principles of Awful Interface Design

- Design: don't format output. Put data out without regard to making it legible or attractive
- Principle: do what is easy for the designer, not the user

- Design: make the user's sequence of inputs and corresponding outputs as unorganized and chaotic as possible
- Principle: users love puzzles

- Design: don't give the user any control over how the program's computation proceeds. Especially don't provide any means of correcting input or stopping processing of garbage
- Principle: your design is perfect. Why should the user have any control

- Design: give the user choices but make sure that it is not clear what happens when a selection is made
- Principle: I know how it works. Why can't the user figure it out?

End of Principles

Organizing Principles

- Organizing principles are those design ideas, data structures, operation sequences, and partitions that make the design appear to be simple.
- Organizing principles often are discovered by thinking about:
  - key program events
  - special data structures
  - the smallest set of information necessary to complete a task
  - designing a language to represent the program's critical processing
- Developing organizing principles is the most creative part of design.
- You know you have a good set of organizing principles when you are sure that you can successfully implement the program even though you have not yet solved all its problems.
- The architecture of your program is simply its organizing principles and partitioning.

Program Structure

- Structure is determined by the modules you use, the classes which populate the modules, and relationships between classes.
- One module should be an executive and all others servers.
  - each server should do one thing well
  - servers may use services of other modules but should usually hide that from the client
  - modules can be identified by listing all the activities of your program, taking each activity to be a module candidate.
- Classes are also servers.
  - each class should provide a single type of service to its clients.
  - classes are found in the problem domain as things a user sees and interacts with, e.g., screens, dates, directories, graphs, ...
  - classes are also found in the solution domain as things the designer uses to implement programs, e.g., lists, maps, strings, ...

Inventing a Structure

- How do I develop a program's structure?
  - List its principle activities and the information needed to sustain each activity. Data flow diagrams and activity charts are good for this.
  - Each process in a dataflow diagram is a candidate to become a module or class component
  - Think about what information each component needs to generate or transform and pass on to its fellows. The information may become data members and transformations may become member functions.
  - Invent data structures in which to collect information, store it, and disclose it to clients. Each data structure is a candidate for a class.
  - look vigorously for organizing principles -- those bright ideas which simplify, make elegant, or make powerful the processes you implement
  - capture your structure with OMT diagrams and struc-ture charts

Communication

- How do I establish effective communication?
  - Do as little as possible by partitioning processes into sets which don't have to exchange much data.
  - Make transferred data volume as small as possible by condensing before transmission.
  - Pass data references rather than data itself where that makes sense.
  - Use the command line, files, and user console for data exchange where each are most suitable:
    - command line for setting the mode or format of computations with a few words.
    - files as sources and sinks of large complex bodies of information.
    - use run-time queries for data and control that is unique each time a program is run. Minimize the amount of data requested of the user at run-time.
  - Describe data transactions by some logical model, then create classes to represent the transmitter and/or receiver.

Ownership

- How do I establish effective ownership?
  - A component which needs information should create it if possible, own it, and dispose of it when done.
  - If only one class needs the services of another, think about making the needy class the owner of an object of the serving class. That way other components in the program don't have to be aware of messages sent between needy and server.
  - If a class you are designing needs the services of an object of an other class and the object already exists for some reason (perhaps only some other component has the information necessary to build it) then think about making a reference to the object a private member of your class. That way it's clear your class does not own the object and your class does not suffer the performance penalty of copying another object.
  - When objects are created, try very hard to have the creator also dispose of the objects. Otherwise owner-ship responsibilities are shared between the creator and disposer. That makes the design more complex and much harder to make correct.
  - Containers should be designed to own their objects, e.g., they use value semantics. If you need reference semantics you simply make the type of the contained object a pointer to the object.

Visibility

- How do I control visibility?
  - One of the most important design principles is to minimize the need to know. One component should have to know as little as possible about another. That way when a change has to be made in a component it is likely not to affect other components, except possibly its owner.
  - To hide a component we simply make another component own it and keep it private.
  - Another aspect of need-to-know is implicit assumptions we make about the way another component behaves. This assumption coupling causes a lot of problems when we integrate software developed separately in time or place. It also makes debugging much more difficult.
  - We should strive to minimize the number of assumptions our clients have to make about our components by making their public interfaces simple and logical.
  - We should make explicit, on the manual page and in function prologues, the assumptions we are forced to make in the design of our classes. That way our clients don't get ugly surprises when they try to use our components.

Modules

- Every program is composed entirely of modules
- Every server module is composed of
  - header file defining public interface. It contains:
    - compilation directives preventing multiple compilation
    - manual page supporting reuse by disclosing platform, author, purpose, and operations
    - maintenance page supporting reuse by disclosing build process and bug fixes
    - declarations announcing module services
  - implementation file defining implementation. It contains:
    - prologue identifying author and matching header
    - function and data definitions
    - test stub supporting incremental development
- Modules should not put non-constant data or pointers in public interface.
- All functions which support the module but are not needed by clients should not be in the public interface. Make them private or protected members of a class or type them as static and don't declare them in header.
- Modules should be cohesive, e.g., focused on a single activity or providing a single service.

Design Notes for Modules

- ***Encapsulation***
  - If global variables or functions are private to a module, declare them static. Only use const global data.
  - Use reference types to give read and write access to internals. Don't return pointers.
- ***Initialization Efficiency***
  - static arrays are useful for local initialized arrays (a keyword table should be local, but only initial'd once, so make it static)
- ***Constantness***
  - if function does not change arg declare arg const
  - If member function does not change object state declare it const (put const after parenthesis and before open brace).
- ***Errors***
  - avoid returning references which may become invalid. return reference only if object existed before function call
  - when handling errors in functions that return references, if error return reference to static dummy value or throw exception.
  - avoid comparing signed with unsigned values; if you compare signed with unsigned the compiler will treat the signed value as unsigned possibly causing suprises.
  - avoid recursive functions which are likely to be called many times - possible stack overflow
  - Declare const pointers correctly, e.g.:

    const int *pint = &i declares non-const ptr to const int

    int* const pint = &i declares const ptr to non-const int

    const int* const pint = &i declares const ptr to const int

What is an Object?

- What is an Object?
  - An encapsulated software component that supports a simple logical model through the names and behaviors of its public interface members. An object announces its logical model and public interface details on its manual page.
  - A component which is self sufficient. It requires no support from clients to provide for its operation other than to send it messages and receive its messages.
    - It allocates whatever system resources it needs when it is created and releases them when it is destroyed.
    - An object makes as few assumptions about its external environment as possible, and when assumptions are necessary, it either follows a traditional pattern or notifies clients in its manual page of these special assumptions.
    - If it modifies its external environment during its operations it restores the environment as soon as those operations conclude.
    - It handles its own errors and, when appropriate, notifies its external environment about its error status.
  - An object is often an instance of a type, that is, a data structure and a set of allowable operations for that data.

Object Oriented Design

- What is object oriented design?
  - Almost all effective design approaches begin by partitioning a program's requirements into modules, each of which focuses on a single activity.
  - Object oriented design further partitions each module into a set of interacting objects which carry out their activities by sending each other messages.
  - Object oriented design partitions need-to-know through the use of class hierarchies. A base class establishes a protocol language for all clients to use. Derived classes implement the protocol in ways tailored to each derived type. The client stays blissfully ignorant of the details which distinguish one derived type from another.
  - Clients use the protocol through base class pointers or references which are set to one or another of the derived class objects. The polymorphic action of dynamic binding allows each object to determine the way it responds to a client message passed through a reference or pointer to itself.
  - Often a client does not need to know which object is being used. Clients view all objects as base class objects.
  - Protocol classes, command objects, and finite state machines are all examples of partitioning need to know through the use of class hierarchies.

Object Oriented Design Strategies

- All OO designs use one or more of the following three strategies:
  - Encapsulate processing in one or more classes, carefully separating into simple, logical, public interfaces and private or protected implementations. This division and encapsulation tends to make each class relatively independent of other processing, simplifying design, implementation, and debugging.
  - Extend existing classes using inheritance. Use base classes to represent a complete but small and simple logical model, e.g., vector, string, … Use inheritance to specialize base class behaviors. The resulting derived classes just modify base class behavior by overriding its virtual member functions or add to its behavior by adding new member functions in their public interfaces.
  - Loosely couple client code to server classes with polymorphic relationships. Polymorphism is achieved by defining a language for clients to use in a base class, then supplying clients with derived class objects which support the language in ways unique to each object. The client can often be unaware of which type of derived object it interacts with since it uses only the base class protocol. Each object is responsible for treating the protocol in ways appropriate for its type. This is a very effective way to limit client code's need to know processing details which distinguish one type of derived object from another.

Good Neighbor Policy

- When designing programs remember that you do not have permanent exclusive rights to:
  - memory
  - disk space
  - screen formats
  - I/O channel formats
  - peripherals
  - public names
  - files
  - database locks
  - operating system semaphores
  - communication lines
  - environment variables

- Release system resources as soon as you can. Don't use more than you need.
- Assign memory and files dynamically when you need them and release as soon as you are done.
- If you change iostream formats save the original formats and restore quickly. Do the same for screen formats and environment variables.
- Use namespaces, classes, and structures to reduce the number of names you place in the public domain.

Good Housekeeping Policy

- Software components should be self sufficient:
- Do:
  - Manage components resources within component
    - Memory
    - Mutexes and critical sections
    - Database locks
  - Handle component errors
    - Validate inputs
    - Catch exceptions
  - Provide simple default decisions
    - Common input values
    - Wildcard expansion
    - Path handling
- Don't:
  - Specify explicit pathnames
  - Share resource management with users
  - Abort processing

Design Notes for Classes

References:

- Effective C++, Scott Meyers, Addison-Wesley, 1992

- ***Special member functions:***
  - compiler will generate public copy ctor, assignment operator, destructor, and address-of operators (const and non-const)
    - if you don't declare any ctors the compiler will generate a default (void) ctor
    - if this is a derived class and the base had a destructor the compiler will generator a destructor

    all of the above are generated only if needed by code
  - explicitly disallow use of implicitly generated member functions you don't want by declaring them private
  - always define a copy constructor, assignment operator, and destructor for classes which allocate system resources to objects
  - assignment operator for a derived class should call assignment for its base class to assign the base part
  - the =, (), [], and -> operators must be member functions
- ***Constantness:***
  - if member function does not change state of object, declare function const
  - declare method arguments as constant references whenever you can
- ***Static members:***
  - static member data is shared by all objects
  - static member functions are independent of objects, can access only static or global data, and are qualified by class name

Design Notes for Classes

- ***Encapsulation: classes manage their own data:***
  - avoid data members in public interface
  - avoid returning handles to internal data from const member functions
  - member functions returning references give read and write access to member data.
  - avoid non-const member functions that return pointers or references to members less accessible than themselves
- ***Composition:***
  - initialize bases and data members with explicit calls to constructors in an initialization sequence for each constructor you write.
  - don't use array declarations in classes. Instead, use pointers to arrays on the heap. It's then easy to write resize() functions to make your objects expandable and the objects are smaller so occupy less stack.
- ***Frequent Errors:***
  - never return a reference to a local object or a dereferenced pointer initialized by new within the function
  - use same form in corresponding calls to new and delete
  - don't check return value of new – be prepared for an exception
- ***Type coercion:***
  - write promotion constructors for all types your class needs to convert from, but try to write cast operators only for primitive types and library types
  - a compiler ambiguity can occur if your write a promotion constructor x(Y&) and a cast of Y to X in Y
  - when ambiguity does occur you can resolve by explicitly calling a conversion
  - explicit keyword forces a promotion constructor to be call explicitly.

Design Notes for Classes

- ***Efficiency:***
  - pass and return objects by reference instead of by value where it makes sense to do so
- ***Initialization:***
  - prefer initialization to assignment in constructors:
    - const and reference members must be intialized.
    - construction proceeds in two phases:
      - initialization of data members in the order of their declaration in the class
      - execution of the body of the constructor called
    - if you don't initialize an element its default constructor is called, then you have to assign to it
    - if you use an initialization for that element only the copy constructor is called
  - list non-static members in an initialization list in the order in which they are declared
    - members are initialized in the order they are declared regardless of the order specified in an initialization list
    - to avoid surprises use the same order
  - static members are intialized once, before main() entry. They must be defined globally in implementation file.
- ***Design alternatives:***
  - choose between overloading and default parameters
  - functions can be grouped into an all static class
  - return value of overloaded operator need not be a class object

Assignment

- Have operator= return a reference to *this.
- Assign to all data members in operator=(...).
  - default assignment operator performs member-wise assignment on all data members
  - there is no way to selectively overide the default assignments - either accept the default or take over assignments to all members
  - adding a new data member usually means that both the copy ctor and assignment operator need to be updated
  - since assignment operator is not inherited, a derived class assignment must take care of both base and derived elements - do that as follows:

```
class A { ... };  class B:public A { ... }

B& B::operator=(const B& b) {
  if (this == &b) return *this;
  ((A&)*this) = b;  // call operator= on A part
      ... now assign B data elements from b
  return *this;
}
```

  You may replace the cast `(A&)*this` with the operator form: `A::operator=(b)` if you wish.

- Check for assignment to self in operator=, as shown above.

Member Pointers

- ***Member pointers:***
  - an ordinary pointer can be used to access a member of one specific object.
  - member pointers can be used to access a member of any instance of a given class
  - to dereference a member pointer you have to specify which object you want, e.g.:

```
struct X { int a, b, c; };
X x, y;

int X::*ptr = &X::a;
int u = x.*ptr;   // access to x.a
int v = y.*ptr;   // access to y.a
```

  - to point to a member function:

```
class X { public:  int mf( ); ... };

int (X::*fptr)( ) = &X::mf;

X x;
X *y = &x;
(x.*fptr)( );     // x.mf( ) called by object x
(y->*fptr)( );    // x.mf( ) called through
                  //         pointer to x
```

  - a callback to a member function can be implemented using member pointers as above, using the technique discussed in the call back note

Inheritance

- ***Semantic models:***
  - make sure public inheritance models "is-a"
  - model "like-a" by factoring out a common base class
  - model "has-a" or "is-implemented by" through composition or, if you have to, with private inheritance
- ***Accessibility:***
  - you can selectively allow access when deriving privately
  - you can not selectively remove access when deriving publically
  - constructors, the destructor, the assignment and address-of operators are not inherited but defaults are supplied by the compiler (may not be what you want)
  - declaring a constructor with arguments hides the compiler generated default => derived classes must define default constructors
  - copy constructor is not hidden when you declare other constructors
- ***Errors that affect polymorphism:***
  - never redefine an inherited nonvirtual function
  - Don't overload virtual functions.
  - Don't use default parameters with virtual functions.
  - Don't attempt to overload base class members in a derived class.
  - avoid casts down an inheritance hierarchy
  - make destructors virtual in base classes

Inheritance (continued)

- ***Abstract classes:***
  - a pure virtual function can be defined
    - define body just like any other member function
    - To inherit in a derived class just provide the overriding declaration but provide no implementation.
    - can be called by member functions or member functions of a derived class to do common processing

Data Members

Data members of a class you design are either:

- ***Built-in types:***
  need no special treatment.
- ***Objects of another class***:
  Your class owns these members and is responsible for their construction and destruction. Your constructors should initialize these with an initialization sequence if they need construction with arguments.
- ***Pointers***:
  Your class owns the objects pointed to and is responsible for constructing and deleting them on the heap. If you need to refer to an object you don't own try not to use pointers. Use references instead.
- ***References***:
  Your class does not own the objects referred to. You must initialize your member references using an initialization sequence naming existing objects.

  Using references for data members you don't own does two things for you. It avoids the overhead associated with copying or constructing the object and it makes it very clear to someone that has to maintain your code (maybe you in three months) that the class does not own the object.

Classes Summary

- Place one class or a few logically related small classes in a single module.
- Each class has a public interface defined by its members in the public section of the class declaration.
- Each class has a private implementation defined by the class's private or protected sections.
- Don't put non-const data or pointers in the class's public interface. Put them in the its private implementation.
- Don't declare arrays in a class. Use pointers to memory allocated on the heap. This is much more flexible since the class can resize a dynamic array if needed. This also avoids possible stack overflows due to need for large stack frames to hold arrays.
- Always check for allocation success, when using new, by catching allocation exceptions. Another, less used, alternative is to use the function set_new_handler which makes new call an error handler if the allocation fails.
- If a class has any pointers as data members it should provide a copy constructor. assignment operator, and destructor. The compiler generated copy and assignment do only shallow copies and assignments of pointers. Compiler generated destructors do only member-wise destruction, which does nothing for pointers.
- If you don't think your class semantics should support copying or assignment you can make those members private so clients can't get access.

Comments

- Don’t include *.cpp files. That makes the compiler handle more code at one time which, if the program is large, may cause compilation failure. Also that forces the included file to be recompiled even if it has not changed.

- Don’t use array declarations in class declarations. Use pointers to arrays on the heap.
  - Declaring arrays make the object bigger than necessary. This may cause stack overflow.
  - Using pointers to heap allocations is more flexible since the size of the array can be changed at run time by reallocating and copying the contents.

End of Presentation