CSE687 – Object Oriented Design Class Notes

Design Guidelines

Jim Fawcett

Spring 2015

Excerpts from and addendums to:

"Enough Rope to Shoot Yourself in the Foot",

Allen Holub, McGraw-Hill, 1995

Prime Directive

- ***No surprises***
  - A component, e.g., a package or class should act the way it looks like it should act.
  - The interface should describe what it does in a way that any competent developer can understand.
- ***Maximize Cohesion***
  - Things that are grouped together should be related in function and be focused on a single objective.
- ***Minimize Coupling***
  - When a component changes, everything it's coupled to may need to change.
  - Try to couple only to interface, not implementation.
  - Try to minimize "assumption" coupling and "need to know" coupling as well as data coupling.

Decide in Haste
Repent at Leisure

Kiss Principle

- ***Keep It Small and Simple***
  - Don't solve problems that don't yet exist.
  - Solve the specific problem, not the general case
    - but don't make it needlessly inflexible either
  - Keep the door open for extension through composition and inheritance
  - Use polymorphism to encapsulate "need to know" in specific derived classes, allowing clients to be blissfully ignorant, knowing only the base class protocol.
  - Design function code so that it:
    - fits on a single page
    - has cyclomatic complexity well below 10
  - Keep a package small enough that its structure chart fits on a single page

Separate Interface from Implementation

- Use encapsulation to force clients to program to your interface, not your implementation.
- Hide any complex design details inside your implementation
- Make your interface simple and as small as is practical.
- Don’t return non-constant pointers in public class interfaces:
  - makes clients need to know your implementation
  - Creational functions are an exception to this rule
- For classes, use private or protected keywords:
  - qualify all data as private or protected
  - qualify as private or protected any methods that are complex or dangerous for client use
- Declare and implement global functions and classes that are not intended for client use in the implementation file (don’t declare in header).

Decompose into Smaller Tasks

- Break a complex operation into smaller simpler pieces.
  - if you can't say it well in English (Hindi, Mandarin, …)
    you can't say it well in C++
  - The act of writing out a description of what a program does, and what each component does, is a critical step in the thinking process, even if the result is just one or two pages.
  - If you can't write it clearly then you probably haven't fully thought out either the problem or its solution.
  - When you're done, you have a specification - the only reasonable basis for testing.
- Design is a decomposition process in the application domain.
- Implementation is a re-composition process in the solution domain.

Small is Beautiful

- Large tasks are unmanageable unless they are broken down into small cohesive subtasks.
  - We emphasize use of packages to compose a large program.

- Sometimes large tasks are best accomplished by a collection of small modular programs that use a common representation:
  - executing tasks can be combined in flexible ways
    - use the right tool for each specific job
    - use parts of the collection in ways the designer never thought of
  - new tools are easily added as the tasks and goals evolve
    - UNIX tool set
    - control system computer aided design and analysis
  - new uses often are found if the tools are flexible and easy to use

User Interface Should be Transparent

- Don’t let easy to learn translate into ackward to use.
- Interfaces shouldn't look like computers, they should look like solutions to a task.
- The fastest editor I ever used was the RT11 TECO editor.
  - It was a line editor, not based on a GUI
  - It was brutally hard to learn because it used control keys for all commands
  - Once you learned it, ***NOTHING*** interrupted your typing. You didn’t have to stop and grab a mouse every third sentence.
  - After a few months of use it became invisible. There was nothing conscious between you and the words flowing out on the screen.
- Measure productivity in the number of keystrokes it takes to complete a task.

Read Code

- Read a lot of code.
  - You learn by seeing how others write code. Look at as many samples of good code as you can.
  - Look critically at your own code.
  - Read several of the better trade journals, e.g., C++ report, C/C++ User's Journal, IEEE Computer Magazine, IEEE Software Magazine.
- Write a lot of code.
  - When you're starting a big job, write small prototypes to try out your ideas and be prepared to throw them away or rebuild them before launching the final construction.
  - Use an editor's red pencil on your code. Strike out unnecessary code, simplify, reword, repartition, until you're reasonably satisfied.
  - Be prepared to throw the first one away.

Write for Maintenance

- The maintenance programmer is you!
- Any software that is useful is written once, but read many times.
  - a lot less effort is expended over the lifetime of the program if the designer takes the time to document, design, and implement carefully
- You will spend far more time reading your code than writing it.
  - as you build a package, the first functions built are re-read many times as you build later functions that depend on them.
  - careful unit test of a package will probably take more time than its initial construction but save a lot of debugging time downstream.
- Others will read your code to understand when, where, and how to use it.

Performance is very Important, But...

- Less important than correctness:
  - no point in generating errors very quickly
- Less important than robustness:
  - no one will trust your code if it crashes often
- Less important than maintainability:
  - as soon as a program is put into service, if it's useful, users want more functionality.
  - adding new features to unmaintainable code takes us back to the first two points
- Less important than reusability:
  - we won't be in business very long if we're not as productive as our competitors.
  - in a labor intensive business like software development, that means reuse

Formatting and Documentation

- Software should be self-describing:
  - Unlike most other engineering disciplines, software has the ability, if well written, to capture, store, and disclose on demand, the technology used for its construction.
  - if you use specialized algorithms or technology place citations to references so others can understand how your code works.

- Uncommented code has no value:
  - uncommented code is unmaintainable
  - manual and maintenance information should accompany every package
  - most functions should have a (brief) prologue - perhaps only a single line - and comments only to describe any subtle code.

Documentation Style

- Let code describe that which code describes best.

- Reduce clutter:
  - make comments as brief as possible
  - don’t put descriptive comments in class declarations, save them for member function definitions
  - don’t put inline functions inside class declarations.
    - put very simple functions (one or two lines) in the header file just after their class declaration and use the inline keyword
    - Put all the rest in implementation file unless they are templatized. Templatized functions you put in the header file without inline keyword.

Diagrams

- Use diagrams in requirements and design documents:
  - data flow diagrams to describe the basic abstractions flows
  - class diagrams to describe the static logical structure
  - event trace and activity diagrams to describe dynamic behavior
  - structure charts to show calling relationships
    - always provide one per module if there is significant function layering
  - data structure diagrams show the organization of your data

- Words are much less effective without diagrams.

- A diagram may be worth a thousand words, but **only if** it is accompanied by a paragraph or two of discussion.

Comments

- Don’t comment the obvious.
- Do put comments where they are needed:
  - Once per package:
    - ***Manual Page***
      - Briefly state purpose, operation, and public interface.
    - ***Maintenance Page***
      - Briefly list maintenance history and state build process including file dependencies
  - Once per file:
    - provide prologue: state name of file, brief phrase describing contents, state language, platform, application, and author
  - Once per function:
    - state action
    - discuss inputs and outputs only if type and format are not obvious
    - put brief comments in code only if semantics are not obvious

White Space is Important

- Show scope level with indentation.
- Set editor to replace tabs with spaces
  - you want tabs to be three or four spaces
  - every printer on earth will make them eight spaces unless it is programmed to do otherwise
- Use page breaks between functions that would otherwise be split across pages:
  - if your editor does not support page breaks, e.g., VC++, you can create one from the command line by copying a ^L from the key-board to a file:

    ```
    copy con >ff
    ^L^Z
    ```

    Then load the ff file into the editor, copy its contents, and paste it, inside a comment, wherever you need it.

Names are Important

- Well chosen names make code nearly self documenting.
  - names should be common words, describing what the file, class, function, argument, or variable does.
  - use one character names only for indices declared, defined, and used locally
  - use names just long enough to be descriptive.
  - use a consistent style of separation, e.g.:
    severalWordName vs. several_word_name
  - use aliases and typedefs sparingly
    - if typedefs are exported as part of the public interface, then describe them in the user documentation included in the module.
    - avoid routinely redefining standard types

Data Types are Important

- Don’t use global data except for constants that should be universally known throughout a package.
  - global names shared between components destroy their reusability
  - non-constant global data makes code maintenance very difficult
- Don’t return non-constant pointers as part of a public interface
  - they give access to memory, not objects
  - clients have to understand your design to use them properly
- It is acceptable to return a reference to a well-designed object:
  - the reference provides access to object only through its interface
  - if you do, the object type referred to should be described in the documentation of your user interface
- Minimize use of static data and try not to use global data at all.
  - both cause problems in recursive and multi-threaded code

Minimize Dependencies

- Don’t make unnecessary dependencies
  - only include header files that are needed ***in the file where included***
  - program to abstract interfaces wherever that makes sense
    - that minimizes compile-time dependencies and need-to-know
  - never declare using namespace statements in header files
    - that declares the using statement in any client’s code that includes your header file
  - try very hard not to require preconditions for clients to use your code
    - when you have to, make sure the conditions are documented as part of your user interface
    - silent assumptions by one component about the behavior of another component cause a lot of grief during integration and maintenance of your code

Handling Errors

- Test routines should not be interactive.
  - a non-interactive test routine can be exhaustive
  - users providing inputs will not be nearly as complete
- Every package should have a test stub to implement construction tests.
- An error message should help a user fix the error.
- Don’t display error messages if your code can recover.
- It is often very useful to provide error trace functions that are easily adapted to different environments:
  - use synchronization of output streams in multi-threaded code
  - use message boxes in GUI applications
  - use streams which can be standard I/O or logging files
- Always flush the output stream if more than one thread share the same stream.

Handling Pointers

- Always initialize a pointer close to its declaration:
  - avoids use of a pointer you ***assumed*** was initialized but wasn't
- If a function you use has an argument that points to a result ***you must know:***
  - ***has the function allocated storage or do you?***
  - ***is the storage heap memory? If so you must deallocate.***
  - ***if you supply the content for that output, is the allocated storage large enough?***
  - strcat, strcpy, strdup are very common sources of pointer errors
- Don't pass around non-const pointers:
  - that forces clients to know your design:
    - is the pointer initialized?
    - what is its valid range?
    - does the client call free or delete on that pointer?

Handling Pointers Again

- Be careful incrementing pointers into an array. Incrementing and assignment statements are valid only from the first element to one past the last element:

```
int array[SIZE];
int *p = array+SIZE;        // ok, can go one past end
while(--p >= array)         // may not work, language
                            // doesn't support going below
                            // base address
```

Architecting and Designing in C++

- Use diagrams to think about classes and class relationships before you write code:
  - UML class diagrams show class relationships
  - structure charts show complex method layering
  - event trace diagrams document evolution of program messages and events
- Use data flow diagrams to work out partitioning strategies.
- Use diagrams to think about data structures.

Class Structure

- Choose composition over derivation for reuse.
- Use inheritance and polymorphism to define a protocol language:
  - clients of the class hierarchy need only know the protocol, not the derived class details
  - use protocol language to build reusable components that need not know any application details
  - use protocol to provide a receptacle for any of a set of components which may be extended at some later time
- Do not provide public access to private data.
- Don’t put function bodies in class declarations:
  - put inline definitions and template definitions after class declaration

Avoiding Pitfalls

- Return by value objects that don't exist before a function call.
- Pass and return by reference when you can.
- Prefer const references as function inputs.
- Constructors with arguments should always use initialization sequences.
  - Derived class constructors should always explicitly initialize their base classes and member objects.
  - Derived class copy constructors ***must*** use an initialization sequence to call their base copy constructor.
- Assignment in a derived class should use the base' assignment operation to get the base part assigned.
- Don't call virtual functions in a constructor for the same class.
- Make destructors virtual for any class that may serve as a base.

Overloading Operators

- Define:
  - operator+, operator-, operator*, and operator/

  in terms of :
  - operator+=, operator-=, operator*=, and operator/=

- Remember the binary operator model:
  - operators as class members: x@y ⇔ x.operator@(y)
  - operators as global functions: x@Y ⇔ operator(x,y)

Use the Whole Language

- Understand all the major features of the language:
  - classes
  - composition
  - inheritance
  - polymorphism
  - templates
  - exceptions
  - standard library
- Study Design Patterns to see smart, tested ways of using OOD.
  - “Design Patterns”, Gamma et. al., Addison-Wesley, 1995
- Then use the appropriate tool for the job.
  - not every design needs all of the language or sophisticated patterns, but every feature and pattern has problems that they solve better than other know ways.

Look at Other Languages

- Other OOD languages:
  - C# and Java: designed to be used in a distributed environment
  - Eiffel: provides direct support for Design-by-Contract
- Scripting languages:
  - JavaScript, VBScript:
    - languages embeddable in html, making active web pages
  - Perl, Tcl, Python, Ruby:
    - designed to be integration languages
- Functional languages:
  - ml, lisp, mathematica:
    - have been used for prototyping and knowledge representation
- Declarative languages:
  - Prolog, Leda
    - used for expert systems, theorem proving

SW Development is a Service Industry

- Ask people what they want, then do what they tell you.
- What's the point of building a program no one wants?
- ***Designers*** need to talk with the ***end users***.
  - Big Government job?
    - There is always on-site installation and customer maintenance.
  - Commercial shrink-wrap product?
    - talk to users of the previous version
  - Embedded software?
    - talk to the production engineers and installers on the factory floor
- Make an end-user part of the development team.
- If you're designing development tools, use them yourself, while you are developing them.

End of Presentation