The following list is essentially the table of contents for my book *Enough Rope to Shoot Yourself in the Foot* (McGraw-Hill, 1995). The book was written with C/C++ in mind, but most of the rules apply to programming in general and OO programming in other languages (such as Java) in particular. You should go get the book if you want to see why the rules are what they are and to see detailed explanations for each rule. Bear in mind, though, that all of these are just rules of thumb. There are always exceptions.

The Design Process

1 The essentials of programming: No surprises, minimize coupling, and maximize cohesion
2 Stamp out the demons of complexity (Part 1)

- 2.1 Don't solve problems that don't exist
- 2.2 Solve the specific problem, not the general case

3 A user interface should not look like a computer program (the transparency principle)
4 Don't confuse ease of learning with ease of use
5 Productivity can be measured in the number of keystrokes
6 If you can't say it in English, you can't say it in C/C++
6.1 Do the comments first
7 Read code

- 7.1 There's no room for prima donnas in a contemporary programming shop

8 Decompose complex problems into smaller tasks
9 Use the whole language (Use the appropriate tool for the job)
10 A problem must be thought through before it can be solved
11 Computer programming is a service industry
12 Involve users in the development process
13 The customer is always right
14 Small is Beautiful. (Big == slow)

General Development Issues

15 First, do no harm
16 Edit your code
17 A program must be written at least twice
18 You can't measure productivity by volume
19 You can't program in isolation
20 Goof off
21 Write code with maintenance in mind—the maintenance programmer is you

21.1 Efficiency is often a bugaboo

Formatting and Documentation

22 Uncommented code has no value
23 Put the code and the documentation in the same place
24 Comments should be sentences
25 Run your code through a spelling checker
26 A comment shouldn't restate the obvious
27 A comment should provide only information needed for maintenance
28 Comments should be in blocks
29 Comments should align vertically
30 Use neat columns as much as possible
31 Don't put comments between the function name and the open brace
32 Mark the ends of long compound statements with something reasonable
33 Put only one statement per line
34 Put argument names in function prototypes
35 Use a “predicate” form to split up long expressions
36 A subroutine should fit on a screen
37 All code should be printable
38 Use lines of dashes for visual separation between subroutines
39 White space is one of the most effective comments
40 Use four-space indents
41 Indent statements associated with a flow-control statement

41.1.Comments should be at the same indent level as the surrounding code

42 Align braces vertically at the outer level
43 Use braces when more than one line is present under a flow-control statement

Names and Identifiers

44 Names should be common English words, descriptive of what the function, argument, or variable does

44.1.Do not clutter names with gibberish

45 Macro names should be ENTIRELY_CAPITALIZED

45.1 Do not capitalize members of an enum
45.2 Do not capitalize type names created with a typedef

46 Avoid the ANSI C name space
47 Avoid the Microsoft name space
48 Avoid unnecessary symbols
49 Symbolic constants for Boolean values are rarely necessary

Rules for General Programming

50 Don't confuse familiarity with readability

51 A function should do only one thing
52 Too many levels of abstraction or encapsulation are as bad as too few
53 A function should be called more than once, but…

53.1 Code used more than once should be put into a function

54 A function should have only one exit point

54.1 Always put a return at the outer level

55 Avoid duplication of effort
56 Don't corrupt the global name space

56.1 Avoid global symbols
56.2 Never require initialization of a global variable to call a function

56.2.1 Make locals static in recursive functions if the value doesn't span a recursive call 56.3 Use instance counts in place of initialization functions
56.4 If an if ends in return, don't use else

57 Put the shortest clause of an if/else on top
58 Try to move errors from run time to compile time
59 Use C function pointers as selectors
60 Avoid do/while loops

60.1 Never use a do/while for a forever loop

61 Counting loops should count down if possible
62 Don't do the same thing in two ways at the same time
63 Use for if any two of an initialization, test, or increment are present
64 If it doesn't appear in the test, it shouldn't appear in the other parts of for statement
65 Assume that things will go wrong
66 Computers do not know mathematics

66.1 Expect the impossible
66.2 Always check error-return codes

67 Avoid explicit temporary variables
68 No magic numbers
69 Make no assumptions about sizes
70 Beware of casts (C issues)
71 Handle special cases directly
72 Don't try to make lint happy
73 Put memory allocation and deallocation code in the same place
74 Heap memory is expensive
75 Test routines should not be interactive
76 An error message should tell the user what's right
77 Don't print error messages if an error is recoverable
78 Don't use system-dependent functions for error messages

The Preprocessor

79 Everything in a .h file should be used in at least two .c files
80 Use nested #includes

81 You should always be able to replace a macro with a function

- 81.1 ?: is not the same as if/else
- 81.2 Parenthesize macro bodies and arguments

82 enum and const are better than a macro

83 A parameterized-macro argument should not appear more than once on the right-hand side

- 83.1 Never use macros for character constants

84 When all else fails, use the preprocessor

C-Related Rules

85 Stamp out the demons of complexity (Part 2)

- 85.1 Eliminate clutter.
- 85.2 Avoid bitwise masks; use bit fields
- 85.3 Don't use done flags
- 85.4 Assume that your reader knows C
- 85.5 Don't pretend that C supports a Boolean type (#define TRUE)

86 1-bit bit fields should be unsigned

87 Pointers must be above the base address of an array

88 Use pointers instead of array indexes

89 Avoid goto except . . .

OO Programming/Design (C++ and Java)

90 Object-oriented and “structured" designs don't mix

- 90.1 If it's not object-oriented, use C

91 Expect to spend more time in design and less in development

92 C++ class libraries usually can't be used in a naive way

93 Use checklists

94 Messages should exercise capabilities, not request information

95 You usually cannot convert an existing structured program to object-oriented

96 A derived class object is a base-class object

97 Derivation is the process of adding member data and methods

98 Design the objects first

99 Design the hierarchy next, from the bottom up

- 99.1 Base classes should have more than one derived class

100 The capabilities defined in the base class should be used by all derived classes

101 C++ is not Smalltalk—avoid a common object class

102 Mix-ins shouldn't derive from anything in C++, in Java there's no problem if you follow the next rule:

103 Mix-ins should be C++ virtual base classes (in Java they should be interfaces)

104 Initialize virtual base classes with the default constructor

105 Derivation is not appropriate if you never send a base-class message to a derived-class object

106 Choose containment over derivation whenever possible
107 Use private base classes only when you must provide virtual overrides (C++ only)
108 Design the data structures last
109 All data in a class definition must be private
110 Never provide public access to private data
110.1 Do not use get/set functions
111 Give up on C idioms when coding in C++
112 Design with derivation in mind
112.1 A member function should usually use the private data of a class
113 Use const (final in Java)
114 Use struct only if everything's public and there are no member functions (C++ only)
115 Don't put function bodies into class definitions (C++ only)
116 Avoid function overloads and default arguments
117 Avoid friend classes (in Java, don't use package access.)
118 Inheritance is a form of coupling
119 Don't corrupt the global name space

C++ Rules

References

120 Reference arguments should always be const
121 Never use references as outputs, use pointers
122 Do not return references (or pointers) to local variables
123 Do not return references to memory that came from new

Constructors, Destructors, and operator=()

124 Operator=() should return a const reference
125 Assignment to self must work
126 Classes having pointer members should always define a copy constructor and operator=()
127 If you can access an object, it has been initialized
128 Use member-initialization lists
129 Assume that members and base classes are initialized in random order
130 Copy constructors must use member initialization lists
131 Derived classes should usually define a copy constructor and operator=()
132 Constructors not suitable for type conversion should have two or more arguments
133 Use instance counts for class-level initialization
134 Avoid two-part initialization
135 C++ wrappers around existing interfaces rarely work well

Virtual Functions

136 Virtual functions are those functions that you can't write at the base-class level
137 A virtual function isn't virtual when called from a constructor or destructor

138 Do not call pure virtual functions from constructors
139 Destructors should always be virtual
140 Base-class functions that have the same name as derived-class functions generally should be virtual
141 Don't make a function virtual unless you want the derived class to get control of it
142 protected functions should usually be virtual
143 Beware of casts: C++ issues
144 Don't call constructors from operator=()

Operator Overloads

145 An operator is an abbreviation (no surprises)
146 Use operator overloads only to define operations for which there is a C analog (no surprises)
147 Once you overload an operation, you must overload all similar operations
148 Operator overloads should work exactly like they would in C
149 It's best for a binary-operator overload to be an inline alias for a cast
150 Don't go bonkers with type-conversion operators
151 Do all type conversions with constructors if possible

Memory Management

152 Use new/delete rather than malloc()/free()
153 All memory allocated in a constructor should be freed in the destructor
154 Local overloads of new and delete are dangerous

Templates

155 Use inline function templates instead of parameterized macros
156 Always be aware of the size of the expanded template
157 Class templates should usually define derived classes
158 Templates do not replace derivation; they automate it

Exceptions

159 Intend for exceptions not to be caught
160 Throw error objects when possible
161 Throwing exceptions from constructors is tricky

---

http://www.holub.com Allen I. Holub & Associates, Berkeley, CA 510/ 528-2166 [Voice / Fax] (510/ java-166) info@holub.com [email]