Code Complete, 2nd Ed.
Checklists[1]

Steven C. McConnell

[1]Converted to LaTeX by Matthew J. Miller (www.matthewjmiller.net)

Contents

Chapter 3: Measure Twice Cut Once: Upstream Prerequisites

Checklist: Requirements

The requirements checklist contains a list of questions to ask yourself about your project's requirements. This book doesn't tell you how to do good requirements development, and the list won't tell you how to do one either. Use the list as a sanity check at construction time to determine how solid the ground that you're standing on is—where you are on the requirements Richter scale.

Not all of the checklist questions will apply to your project. If you're working on an informal project, you'll find some that you don't even need to think about. You'll find others that you need to think about but don't need to answer formally. If you're working on a large, formal project, however, you may need to consider every one.

Specific Functional Requirements

- ☐ Are all the inputs to the system specified, including their source, accuracy, range of values, and frequency?
- ☐ Are all the outputs from the system specified, including their destination, accuracy, range of values, frequency, and format?
- ☐ Are all output formats specified for web pages, reports, and so on?
- ☐ Are all the external hardware and software interfaces specified?
- ☐ Are all the external communication interfaces specified, including handshaking, error-checking, and communication protocols?
- ☐ Are all the tasks the user wants to perform specified?
- ☐ Is the data used in each task and the data resulting from each task specified?

Specific Non-Functional (Quality) Requirements

- ☐ Is the expected response time, from the user's point of view, specified for all necessary operations?
- ☐ Are other timing considerations specified, such as processing time, data-transfer rate, and system throughput?
- ☐ Is the level of security specified?
- ☐ Is the reliability specified, including the consequences of software failure, the vital information that needs to be protected from failure, and the strategy for error detection and recovery?
- ☐ Is maximum memory specified?
- ☐ Is the maximum storage specified?
- ☐ Is the maintainability of the system specified, including its ability to adapt to changes in specific functionality, changes in the operating environment, and changes in its interfaces with other software?
- ☐ Is the definition of success included? Of failure?

Requirements Quality

- ☐ Are the requirements written in the user's language? Do the users think so?
- ☐ Does each requirement avoid conflicts with other requirements?
- ☐ Are acceptable trade-offs between competing attributes specified—for example, between robustness and correctness?
- ☐ Do the requirements avoid specifying the design?
- ☐ Are the requirements at a fairly consistent level of detail? Should any requirement be specified in more detail? Should any requirement be specified in less detail?
- ☐ Are the requirements clear enough to be turned over to an independent group for construction and still be understood?
- ☐ Is each item relevant to the problem and its solution? Can each item be traced to its origin in the problem environment?
- ☐ Is each requirement testable? Will it be possible for independent testing to determine whether each requirement has been satisfied?
- ☐ Are all possible changes to the requirements specified, including the likelihood of each change?

Requirements Completeness

- ☐ Where information isn't available before development begins, are the areas of incompleteness specified?
- ☐ Are the requirements complete in the sense that if the product satisfies every requirement, it will be acceptable?
- ☐ Are you comfortable with all the requirements? Have you eliminated requirements that are impossible to implement and included just to appease your customer or your boss?

Checklist: Architecture

Here's a list of issues that a good architecture should address. The list isn't intended to be a comprehensive guide to architecture but to be a pragmatic way of evaluating the nutritional content of what you get at the programmer's end of the software food chain. Use this checklist as a starting point for your own checklist. As with the requirements checklist, if you're working on an informal project, you'll find some items that you don't even need to think about. If you're working on a larger project, most of the items will be useful.

Specific Architectural Topics

- ☐ Is the overall organization of the program clear, including a good architectural overview and justification?
- ☐ Are major building blocks well defined, including their areas of responsibility and their interfaces to other building blocks?
- ☐ Are all the functions listed in the requirements covered sensibly, by neither too many nor too few building blocks?

- ☐ Are the most critical classes described and justified?
- ☐ Is the data design described and justified?
- ☐ Is the database organization and content specified?
- ☐ Are all key business rules identified and their impact on the system described?
- ☐ Is a strategy for the user interface design described?
- ☐ Is the user interface modularized so that changes in it won't affect the rest of the program?
- ☐ Is a strategy for handling I/O described and justified?
- ☐ Are resource-use estimates and a strategy for resource management described and justified?
- ☐ Are the architecture's security requirements described?
- ☐ Does the architecture set space and speed budgets for each class, subsystem, or functionality area?
- ☐ Does the architecture describe how scalability will be achieved?
- ☐ Does the architecture address interoperability?
- ☐ Is a strategy for internationalization/localization described?
- ☐ Is a coherent error-handling strategy provided?
- ☐ Is the approach to fault tolerance defined (if any is needed)?
- ☐ Has technical feasibility of all parts of the system been established?
- ☐ Is an approach to overengineering specified?
- ☐ Are necessary buy-vs.-build decisions included?
- ☐ Does the architecture describe how reused code will be made to conform to other architectural objectives?
- ☐ Is the architecture designed to accommodate likely changes?
- ☐ Does the architecture describe how reused code will be made to conform to other architectural objectives?

General Architectural Quality

- ☐ Does the architecture account for all the requirements?
- ☐ Is any part over- or under-architected? Are expectations in this area set out explicitly?
- ☐ Does the whole architecture hang together conceptually?
- ☐ Is the top-level design independent of the machine and language that will be used to implement it?
- ☐ Are the motivations for all major decisions provided?
- ☐ Are you, as a programmer who will implement the system, comfortable with the architecture?

Checklist: Upstream Prerequisites

- ☐ Have you identified the kind of software project you're working on and tailored your approach appropriately?
- ☐ Are the requirements sufficiently well-defined and stable enough to begin construction (see the requirements checklist for details)?
- ☐ Is the architecture sufficiently well defined to begin construction (see the architecture checklist for details)?
- ☐ Have other risks unique to your particular project been addressed, such that construction is not exposed to more risk than necessary?

Chapter 4: Key Construction Decisions

Checklist: Major Construction Practices

The following checklist summarizes the specific practices you should consciously decide to include or exclude during construction. Details of the practices are contained throughout *Code Complete 2nd Ed.*

Coding

- ☐ Have you defined coding conventions for names, comments, and formatting?
- ☐ Have you defined specific coding practices that are implied by the architecture, such as how error conditions will be handled, how security will be addressed, and so on?
- ☐ Have you identified your location on the technology wave and adjusted your approach to match? If necessary, have you identified how you will program into the language rather than being limited by programming in it?

Teamwork

- ☐ Have you defined an integration procedure, that is, have you defined the specific steps a programmer must go through before checking code into the master sources?
- ☐ Will programmers program in pairs, or individually, or some combination of the two?

Quality Assurance

- ☐ Will programmers write test cases for their code before writing the code itself?
- ☐ Will programmers write unit tests for their code regardless of whether they write them first or last?
- ☐ Will programmers step through their code in the debugger before they check it in?
- ☐ Will programmers integration-test their code before they check it in?
- ☐ Will programmers review or inspect each others' code?

Tools

- ☐ Have you selected a revision control tool?
- ☐ Have you selected a language and language version or compiler version?
- ☐ Have you decided whether to allow use of non-standard language features?
- ☐ Have you identified and acquired other tools you'll be using—editor, refactoring tool, debugger, test framework, syntax checker, and so on?

Chapter 5: Design In Construction

Checklist: Design in Construction

Design Practices

- ☐ Have you iterated, selecting the best of several attempts rather than the first attempt?
- ☐ Have you tried decomposing the system in several different ways to see which way will work best?
- ☐ Have you approached the design problem both from the top down and from the bottom up?
- ☐ Have you prototyped risky or unfamiliar parts of the system, creating the absolute minimum amount of throwaway code needed to answer specific questions?
- ☐ Has you design been reviewed, formally or informally, by others?
- ☐ Have you driven the design to the point that its implementation seems obvious?
- ☐ Have you captured your design work using an appropriate technique such as a Wiki, email, flipcharts, digital camera, UML, CRC cards, or comments in the code itself?

Design Goals

- ☐ Does the design adequately address issues that were identified and deferred at the architectural level?
- ☐ Is the design stratified into layers?
- ☐ Are you satisfied with the way the program has been decomposed into subsystems, packages, and classes?
- ☐ Are you satisfied with the way the classes have been decomposed into routines?
- ☐ Are classes designed for minimal interaction with each other?
- ☐ Are classes and subsystems designed so that you can use them in other systems?
- ☐ Will the program be easy to maintain?
- ☐ Is the design lean? Are all of its parts strictly necessary?
- ☐ Does the design use standard techniques and avoid exotic, hard-to-understand elements?
- ☐ Overall, does the design help minimize both accidental and essential complexity?

Chapter 6: Working Classes

Checklist: Class Quality

Abstract Data Types

- ☐ Have you thought of the classes in your program as Abstract Data Types and evaluated their interfaces from that point of view?

Abstraction

- ☐ Does the class have a central purpose?
- ☐ Is the class well named, and does its name describe its central purpose?
- ☐ Does the class's interface present a consistent abstraction?
- ☐ Does the class's interface make obvious how you should use the class?
- ☐ Is the class's interface abstract enough that you don't have to think about how its services are implemented? Can you treat the class as a black box?
- ☐ Are the class's services complete enough that other classes don't have to meddle with its internal data?
- ☐ Has unrelated information been moved out of the class?
- ☐ Have you thought about subdividing the class into component classes, and have you subdivided it as much as you can?
- ☐ Are you preserving the integrity of the class's interface as you modify the class?

Encapsulation

- ☐ Does the class minimize accessibility to its members?
- ☐ Does the class avoid exposing member data?
- ☐ Does the class hide its implementation details from other classes as much as the programming language permits?
- ☐ Does the class avoid making assumptions about its users, including its derived classes?
- ☐ Is the class independent of other classes? Is it loosely coupled?

Inheritance

- ☐ Is inheritance used only to model "is a" relationships?
- ☐ Does the class documentation describe the inheritance strategy?
- ☐ Do derived classes adhere to the Liskov Substitution Principle?
- ☐ Do derived classes avoid "overriding" non overridable routines?
- ☐ Are common interfaces, data, and behavior as high as possible in the inheritance tree?

- ☐ Are inheritance trees fairly shallow?
- ☐ Are all data members in the base class private rather than protected?

Other Implementation Issues

- ☐ Does the class contain about seven data members or fewer?
- ☐ Does the class minimize direct and indirect routine calls to other classes?
- ☐ Does the class collaborate with other classes only to the extent absolutely necessary?
- ☐ Is all member data initialized in the constructor?
- ☐ Is the class designed to be used as deep copies rather than shallow copies unless there's a measured reason to create shallow copies?

Language-Specific Issues

- ☐ Have you investigated the language-specific issues for classes in your specific programming language?

Chapter 7: High Quality Routines

Checklist: High-Quality Routines

Big-Picture Issues

- ☐ Is the reason for creating the routine sufficient?
- ☐ Have all parts of the routine that would benefit from being put into routines of their own been put into routines of their own?
- ☐ Is the routine's name a strong, clear verb-plus-object name for a procedure or a description of the return value for a function?
- ☐ Does the routine's name describe everything the routine does?
- ☐ Have you established naming conventions for common operations?
- ☐ Does the routine have strong, functional cohesion—doing one and only one thing and doing it well?
- ☐ Do the routines have loose coupling—are the routine's connections to other routines small, intimate, visible, and flexible?
- ☐ Is the length of the routine determined naturally by its function and logic, rather than by an artificial coding standard?

Parameter-Passing Issues

- ☐ Does the routine's parameter list, taken as a whole, present a consistent interface abstraction?
- ☐ Are the routine's parameters in a sensible order, including matching the order of parameters in similar routines?
- ☐ Are interface assumptions documented?
- ☐ Does the routine have seven or fewer parameters?
- ☐ Is each input parameter used?
- ☐ Is each output parameter used?
- ☐ Does the routine avoid using input parameters as working variables?
- ☐ If the routine is a function, does it return a valid value under all possible circumstances?

Chapter 8: Defensive Programming

Checklist: Defensive Programming

Too much defensive programming creates problems of its own. If you check data passed as parameters in every conceivable way in every conceivable place, your program will be fat and slow. What's worse, the additional code needed for defensive programming adds complexity to the software. Code installed for defensive programming is not immune to defects, and you're just as likely to find a defect in defensive programming code as in any other code—more likely, if you write the code casually. Think about where you need to be defensive, and set your defensive-programming priorities accordingly.

General

- ☐ Does the routine protect itself from bad input data?
- ☐ Have you used assertions to document assumptions, including preconditions and postconditions?
- ☐ Have assertions been used only to document conditions that should never occur?
- ☐ Does the architecture or high-level design specify a specific set of error handling techniques?
- ☐ Does the architecture or high-level design specify whether error handling should favor robustness or correctness?
- ☐ Have barricades been created to contain the damaging effect of errors and reduce the amount of code that has to be concerned about error processing?
- ☐ Have debugging aids been used in the code?
- ☐ Has information hiding been used to contain the effects of changes so that they won't affect code outside the routine or class that is changed?
- ☐ Have debugging aids been installed in such a way that they can be activated or deactivated without a great deal of fuss?
- ☐ Is the amount of defensive programming code appropriate—neither too much nor too little?
- ☐ Have you used offensive programming techniques to make errors difficult to overlook during development?

Exceptions

- ☐ Has your project defined a standardized approach to exception handling?
- ☐ Have you considered alternatives to using an exception?
- ☐ Is the error handled locally rather than throwing a non-local exception if possible?
- ☐ Does the code avoid throwing exceptions in constructors and destructors?
- ☐ Are all exceptions at the appropriate levels of abstraction for the routines that throw them?
- ☐ Does each exception include all relevant exception background information?
- ☐ Is the code free of empty catch blocks? (Or if an empty catch block truly is appropriate, is it documented?)

Security Issues

- ☐ Does the code that checks for bad input data check for attempted buffer overflows, SQL injection, html injection, integer overflows, and other malicious inputs?
- ☐ Are all error-return codes checked?
- ☐ Are all exceptions caught?
- ☐ Do error messages avoid providing information that would help an attacker break into the system?

Chapter 9: Pseudocode Programming Process

Checklist: The Pseudocode Programming Process

- ☐ Have you checked that the prerequisites have been satisfied?
- ☐ Have you defined the problem that the class will solve?
- ☐ Is the high level design clear enough to give the class and each of its routines a good name?
- ☐ Have you thought about how to test the class and each of its routines?
- ☐ Have you thought about efficiency mainly in terms of stable interfaces and readable implementations, or in terms of meeting resource and speed budgets?
- ☐ Have you checked the standard libraries and other code libraries for applicable routines or components?
- ☐ Have you checked reference books for helpful algorithms?
- ☐ Have you designed each routine using detailed pseudocode?
- ☐ Have you mentally checked the pseudocode? Is it easy to understand?
- ☐ Have you paid attention to warnings that would send you back to design (use of global data, operations that seem better suited to another class or another routine, and so on)?
- ☐ Did you translate the pseudocode to code accurately?
- ☐ Did you apply the PPP recursively, breaking routines into smaller routines when needed?
- ☐ Did you document assumptions as you made them?
- ☐ Did you remove comments that turned out to be redundant?
- ☐ Have you chosen the best of several iterations, rather than merely stopping after your first iteration?
- ☐ Do you thoroughly understand your code? Is it easy to understand?

Chapter 10: General Issues In Using Variables

Checklist: General Considerations In Using Data

Initializing Variables

- ☐ Does each routine check input parameters for validity?
- ☐ Does the code declare variables close to where they're first used?
- ☐ Does the code initialize variables as they're declared, if possible?
- ☐ Does the code initialize variables close to where they're first used, if it isn't possible to declare and initialize them at the same time?
- ☐ Are counters and accumulators initialized properly and, if necessary, reinitialized each time they are used?
- ☐ Are variables reinitialized properly in code that's executed repeatedly?
- ☐ Does the code compile with no warnings from the compiler?
- ☐ If your language uses implicit declarations, have you compensated for the problems they cause?

Other General Issues in Using Data

- ☐ Do all variables have the smallest scope possible?
- ☐ Are references to variables as close together as possible—both from each reference to a variable to the next and in total live time?
- ☐ Do control structures correspond to the data types?
- ☐ Are all the declared variables being used?
- ☐ Are all variables bound at appropriate times, that is, striking a conscious balance between the flexibility of late binding and the increased complexity associated with late binding?
- ☐ Does each variable have one and only one purpose?
- ☐ Is each variable's meaning explicit, with no hidden meanings?

Chapter 11: Power of Variables Names

Checklist: Naming Variables

General Naming Considerations

- ☐ Does the name fully and accurately describe what the variable represents?
- ☐ Does the name refer to the real-world problem rather than to the programming-language solution?
- ☐ Is the name long enough that you don't have to puzzle it out?
- ☐ Are computed-value qualifiers, if any, at the end of the name?
- ☐ Does the name use Count or Index instead of Num?

Naming Specific Kinds Of Data

- ☐ Are loop index names meaningful (something other than i, j, or k if the loop is more than one or two lines long or is nested)?
- ☐ Have all "temporary" variables been renamed to something more meaningful?
- ☐ Are boolean variables named so that their meanings when they're True are clear?
- ☐ Do enumerated-type names include a prefix or suffix that indicates the category—for example, Color_ for Color_Red, Color_Green, Color_Blue, and so on?
- ☐ Are named constants named for the abstract entities they represent rather than the numbers they refer to?

Naming Conventions

- ☐ Does the convention distinguish among local, class, and global data?
- ☐ Does the convention distinguish among type names, named constants, enumerated types, and variables?
- ☐ Does the convention identify input-only parameters to routines in languages that don't enforce them?
- ☐ Is the convention as compatible as possible with standard conventions for the language?
- ☐ Are names formatted for readability?

Short Names

- ☐ Does the code use long names (unless it's necessary to use short ones)?
- ☐ Does the code avoid abbreviations that save only one character?
- ☐ Are all words abbreviated consistently?
- ☐ Are the names pronounceable?
- ☐ Are names that could be mispronounced avoided?
- ☐ Are short names documented in translation tables?

Common Naming Problems: Have You Avoided. . .

☐ . . .names that are misleading?

☐ . . .names with similar meanings?

☐ . . .names that are different by only one or two characters?

☐ . . .names that sound similar?

☐ . . .names that use numerals?

☐ . . .names intentionally misspelled to make them shorter?

☐ . . .names that are commonly misspelled in English?

☐ . . .names that conflict with standard library-routine names or with predefined variable names?

☐ . . .totally arbitrary names?

☐ . . .hard-to-read characters?

Chapter 12: Fundamental Data Types

Checklist: Fundamental Data

Numbers in General

- ☐ Does the code avoid magic numbers?
- ☐ Does the code anticipate divide-by-zero errors?
- ☐ Are type conversions obvious?
- ☐ If variables with two different types are used in the same expression, will the expression be evaluated as you intend it to be?
- ☐ Does the code avoid mixed-type comparisons?
- ☐ Does the program compile with no warnings?

Integers

- ☐ Do expressions that use integer division work the way they're meant to?
- ☐ Do integer expressions avoid integer-overflow problems?

Floating-Point Numbers

- ☐ Does the code avoid additions and subtractions on numbers with greatly different magnitudes?
- ☐ Does the code systematically prevent rounding errors?
- ☐ Does the code avoid comparing floating-point numbers for equality?

Characters and Strings

- ☐ Does the code avoid magic characters and strings?
- ☐ Are references to strings free of off-by-one errors?
- ☐ Does C code treat string pointers and character arrays differently?
- ☐ Does C code follow the convention of declaring strings to be length constant+1?
- ☐ Does C code use arrays of characters rather than pointers, when appropriate?
- ☐ Does C code initialize strings to NULLs to avoid endless strings?
- ☐ Does C code use strncpy() rather than strcpy()? And strncat() and strncmp()?

Boolean Variables

- ☐ Does the program use additional boolean variables to document conditional tests?
- ☐ Does the program use additional boolean variables to simplify conditional tests?

Enumerated Types

- ☐ Does the program use enumerated types instead of named constants for their improved readability, reliability, and modifiability?
- ☐ Does the program use enumerated types instead of boolean variables when a variable's use cannot be completely captured with TRUE and FALSE?
- ☐ Do tests using enumerated types test for invalid values?
- ☐ Is the first entry in an enumerated type reserved for "invalid"?

Named Constants

- ☐ Does the program use named constants for data declarations and loop limits rather than magic numbers?
- ☐ Have named constants been used consistently—not named constants in some places, literals in others?

Arrays

- ☐ Are all array indexes within the bounds of the array?
- ☐ Are array references free of off-by-one errors?
- ☐ Are all subscripts on multidimensional arrays in the correct order?
- ☐ In nested loops, is the correct variable used as the array subscript, avoiding loop-index cross talk?

Creating Types

- ☐ Does the program use a different type for each kind of data that might change?
- ☐ Are type names oriented toward the real-world entities the types represent rather than toward programming-language types?
- ☐ Are the type names descriptive enough to help document data declarations?
- ☐ Have you avoided redefining predefined types?
- ☐ Have you considered creating a new class rather than simply redefining a type?

Chapter 13: Unusual Data Types

Checklist: Considerations In Using Unusual Data Types

Structures

- ☐ Have you used structures instead of naked variables to organize and manipulate groups of related data?
- ☐ Have you considered creating a class as an alternative to using a structure?

Global Data

- ☐ Are all variables local or class-scope unless they absolutely need to be global?
- ☐ Do variable naming conventions differentiate among local, class, and global data?
- ☐ Are all global variables documented?
- ☐ Is the code free of pseudoglobal data-mammoth objects containing a mishmash of data that's passed to every routine?
- ☐ Are access routines used instead of global data?
- ☐ Are access routines and data organized into classes?
- ☐ Do access routines provide a level of abstraction beyond the underlying data-type implementations?
- ☐ Are all related access routines at the same level of abstraction?

Pointers

- ☐ Are pointer operations isolated in routines?
- ☐ Are pointer references valid, or could the pointer be dangling?
- ☐ Does the code check pointers for validity before using them?
- ☐ Is the variable that the pointer references checked for validity before it's used?
- ☐ Are pointers set to NULL after they're freed?
- ☐ Does the code use all the pointer variables needed for the sake of readability?
- ☐ Are pointers in linked lists freed in the right order?
- ☐ Does the program allocate a reserve parachute of memory so that it can shut down gracefully if it runs out of memory?
- ☐ Are pointers used only as a last resort, when no other method is available?

Chapter 14: Organizing Straight Line Code

Checklist: Organizing Straight Line Code

- ☐ Does the code make dependencies among statements obvious?
- ☐ Do the names of routines make dependencies obvious?
- ☐ Do parameters to routines make dependencies obvious?
- ☐ Do comments describe any dependencies that would otherwise be unclear?
- ☐ Have housekeeping variables been used to check for sequential dependencies in critical sections of code?
- ☐ Does the code read from top to bottom?
- ☐ Are related statements grouped together?
- ☐ Have relatively independent groups of statements been moved into their own routines?

Chapter 15: Using Conditionals

Checklist: Conditionals

if-then Statements

- ☐ Is the nominal path through the code clear?
- ☐ Do *if-then* tests branch correctly on equality?
- ☐ Is the *else* clause present and documented?
- ☐ Is the *else* clause correct?
- ☐ Are the *if* and *else* clauses used correctly—not reversed?
- ☐ Does the normal case follow the *if* rather than the *else*?

if-then-else-if Chains

- ☐ Are complicated tests encapsulated in boolean function calls?
- ☐ Are the most common cases tested first?
- ☐ Are all cases covered?
- ☐ Is the *if-then-else-if* chain the best implementation—better than a *case* statement?

case Statements

- ☐ Are cases ordered meaningfully?
- ☐ Are the actions for each case simple-calling other routines if necessary?
- ☐ Does the *case* statement test a real variable, not a phony one that's made up solely to use and abuse the *case* statement?
- ☐ Is the use of the default clause legitimate?
- ☐ Is the default clause used to detect and report unexpected cases?
- ☐ In C, C++, or Java, does the end of each case have a *break*?

Chapter 16: Controlling Loops

Checklist: Loops

Loop Selection and Creation

- ☐ Is a *while* loop used instead of a *for* loop, if appropriate?
- ☐ Was the loop created from the inside out?

Entering the Loop

- ☐ Is the loop entered from the top?
- ☐ Is initialization code directly before the loop?
- ☐ If the loop is an infinite loop or an event loop, is it constructed cleanly rather than using a kludge such as *for i = 1 to 9999*?
- ☐ If the loop is a C++, C, or Java *for* loop, is the loop header reserved for loop-control code?

Inside the Loop

- ☐ Does the loop use { and } or their equivalent to prevent problems arising from improper modifications?
- ☐ Does the loop body have something in it? Is it nonempty?
- ☐ Are housekeeping chores grouped, at either the beginning or the end of the loop?
- ☐ Does the loop perform one and only one function—as a well-defined routine does?
- ☐ Is the loop short enough to view all at once?
- ☐ Is the loop nested to three levels or less?
- ☐ Have long loop contents been moved into their own routine?
- ☐ If the loop is long, is it especially clear?

Loop Indexes

- ☐ If the loop is a *for* loop, does the code inside it avoid monkeying with the loop index?
- ☐ Is a variable used to save important loop-index values rather than using the loop index outside the loop?
- ☐ Is the loop index an ordinal type or an enumerated type—not floating point?
- ☐ Does the loop index have a meaningful name?
- ☐ Does the loop avoid index cross talk?

Exiting the Loop

- ☐ Does the loop end under all possible conditions?
- ☐ Does the loop use safety counters—if you've instituted a safety-counter standard?
- ☐ Is the loop's termination condition obvious?
- ☐ If *break* or *continue* are used, are they correct?

Chapter 17: Unusual Control Structures

Checklist: Unusual Control Structures

return

- ☐ Does each routine use *return* only when necessary?
- ☐ Do *returns* enhance readability?

Recursion

- ☐ Does the recursive routine include code to stop the recursion?
- ☐ Does the routine use a safety counter to guarantee that the routine stops?
- ☐ Is recursion limited to one routine?
- ☐ Is the routine's depth of recursion within the limits imposed by the size of the program's stack?
- ☐ Is recursion the best way to implement the routine? Is it better than simple iteration?

goto

- ☐ Are *gotos* used only as a last resort, and then only to make code more readable and maintainable?
- ☐ If a *goto* is used for the sake of efficiency, has the gain in efficiency been measured and documented?
- ☐ Are *gotos* limited to one label per routine?
- ☐ Do all *gotos* go forward, not backward?
- ☐ Are all *goto* labels used?

Chapter 18: Table Driven Methods

Checklist: Table Driven Methods

- ☐ Have you considered table-driven methods as an alternative to complicated logic?
- ☐ Have you considered table-driven methods as an alternative to complicated inheritance structures?
- ☐ Have you considered storing the table's data externally and reading it at run time so that the data can be modified without changing code?
- ☐ If the table cannot be accessed directly via a straightforward array index (as in the *Age* example), have your put the access-key calculation into a routine rather than duplicating the index calculation in the code?

Chapter 19: General Control Issues

Checklist: Control Structure Issues

- ☐ Do expressions use *True* and *False* rather than *1* and *0*?
- ☐ Are boolean values compared to *True* and *False* implicitly?
- ☐ Are numeric values compared to their test values explicitly?
- ☐ Have expressions been simplified by the addition of new boolean variables and the use of boolean functions and decision tables?
- ☐ Are boolean expressions stated positively?
- ☐ Do pairs of braces balance?
- ☐ Are braces used everywhere they're needed for clarity?
- ☐ Are logical expressions fully parenthesized?
- ☐ Have tests been written in number-line order?
- ☐ Do Java tests uses *a.equals(b)* style instead of $a == b$ when appropriate?
- ☐ Are null statements obvious?
- ☐ Have nested statements been simplified by retesting part of the conditional, converting to *if-then-else* or *case* statements, moving nested code into its own routine, converting to a more object-oriented design, or improved in some other way?
- ☐ If a routine has a decision count of more than 10, is there a good reason for not redesigning it?

Chapter 20: Software Quality Landscape

Checklist: A Quality-Assurance Plan

- ☐ Have you identified specific quality characteristics that are important to your project?
- ☐ Have you made others aware of the projects quality objectives?
- ☐ Have you differentiated between external and internal quality characteristics?
- ☐ Have you thought about the ways in which some characteristics may compete with or complement others?
- ☐ Does your project call for the use of several different error-detection techniques suited to finding several different kinds of errors?
- ☐ Does your project include a plan to take steps to assure software quality during each stage of software development?
- ☐ Is the quality measured in some way so that you can tell whether its improving or degrading?
- ☐ Does management understand that quality assurance incurs additional costs up front in order to save costs later?

Chapter 21: Collaborative Construction

Checklist: Effective Pair Programming

- ☐ Do you have a coding standard to support pair programming that's focused on programming rather than on philosophical coding-style discussions?
- ☐ Are both partners participating actively?
- ☐ Are you avoiding pair programming everything, instead selecting the assignments that will really benefit from pair programming?
- ☐ Are you rotating pair assignments and work assignments regularly?
- ☐ Are the pairs well matched in terms of pace and personality?
- ☐ Is there a team leader to act as the focal point for management and other people outside the project?

Checklist: Effective Inspections

- ☐ Do you have checklists that focus reviewer attention on areas that have been problems in the past?
- ☐ Is the emphasis on defect detection rather than correction?
- ☐ Are inspectors given enough time to prepare before the inspection meeting, and is each one prepared?
- ☐ Does each participant have a distinct role to play?
- ☐ Does the meeting move at a productive rate?
- ☐ Is the meeting limited to two hours?
- ☐ Has the moderator received specific training in conducting inspections?
- ☐ Is data about error types collected at each inspection so that you can tailor future checklists to your organization?
- ☐ Is data about preparation and inspection rates collected so that you can optimize future preparation and inspections?
- ☐ Are the action items assigned at each inspection followed up, either personally by the moderator or with a re-inspection?
- ☐ Does management understand that it should not attend inspection meetings?

Chapter 22: Developer Testing

Checklist: Test Cases

- ☐ Does each requirement that applies to the class or routine have its own test case?
- ☐ Does each element from the design that applies to the class or routine have its own test case?
- ☐ Has each line of code been tested with at least one test case? Has this been verified by computing the minimum number of tests necessary to exercise each line of code?
- ☐ Have all defined-used data-flow paths been tested with at least one test case?
- ☐ Has the code been checked for data-flow patterns that are unlikely to be correct, such as defined-defined, defined-exited, and defined-killed?
- ☐ Has a list of common errors been used to write test cases to detect errors that have occurred frequently in the past?
- ☐ Have all simple boundaries been tested—maximum, minimum, and off-by-one boundaries?
- ☐ Have compound boundaries been tested—that is, combinations of input data that might result in a computed variable thats too small or too large?
- ☐ Do test cases check for the wrong kind of data—for example, a negative number of employees in a payroll program?
- ☐ Are representative, middle-of-the-road values tested?
- ☐ Is the minimum normal configuration tested?
- ☐ Is the maximum normal configuration tested?
- ☐ Is compatibility with old data tested? And are old hardware, old versions of the operating system, and interfaces with old versions of other software tested?
- ☐ Do the test cases make hand-checks easy?

Chapter 23: Debugging

Checklist: Debugging Reminders

Techniques for Finding Defects

- ☐ Use all the data available to make your hypothesis
- ☐ Refine the test cases that produce the error
- ☐ Exercise the code in your unit test suite
- ☐ Use available tools
- ☐ Reproduce the error several different ways
- ☐ Generate more data to generate more hypotheses
- ☐ Use the results of negative tests
- ☐ Brainstorm for possible hypotheses
- ☐ Narrow the suspicious region of the code
- ☐ Be suspicious of classes and routines that have had defects before
- ☐ Check code thats changed recently
- ☐ Expand the suspicious region of the code
- ☐ Integrate incrementally
- ☐ Check for common defects
- ☐ Talk to someone else about the problem
- ☐ Take a break from the problem
- ☐ Set a maximum time for quick and dirty debugging
- ☐ Make a list of brute force techniques, and use them

Techniques for Syntax Errors

- ☐ Don't trust line numbers in compiler messages
- ☐ Don't trust compiler messages
- ☐ Don't trust the compilers second message
- ☐ Divide and conquer
- ☐ Find extra comments and quotation marks

Techniques for Fixing Defects

- ☐ Understand the problem before you fix it
- ☐ Understand the program, not just the problem
- ☐ Confirm the defect diagnosis
- ☐ Relax
- ☐ Save the original source code
- ☐ Fix the problem, not the symptom
- ☐ Change the code only for good reason
- ☐ Make one change at a time
- ☐ Check your fix
- ☐ Look for similar defects

General Approach to Debugging

- ☐ Do you use debugging as an opportunity to learn more about your program, mistakes, code quality, and problem-solving approach?
- ☐ Do you avoid the trial-and-error, superstitious approach to debugging?
- ☐ Do you assume that errors are your fault?
- ☐ Do you use the scientific method to stabilize intermittent errors?
- ☐ Do you use the scientific method to find defects?
- ☐ Rather than using the same approach every time, do you use several different techniques to find defects?
- ☐ Do you verify that the fix is correct?
- ☐ Do you use compiler warning messages, execution profiling, a test framework, scaffolding, and interactive debugging?

Chapter 24: Refactoring

Reasons to Refactor

- ☐ Code is duplicated
- ☐ A routine is too long
- ☐ A loop is too long or too deeply nested
- ☐ A class has poor cohesion
- ☐ A class interface does not provide a consistent level of abstraction
- ☐ A parameter list has too many parameters
- ☐ Changes within a class tend to be compartmentalized
- ☐ Changes require parallel modifications to multiple classes
- ☐ Inheritance hierarchies have to be modified in parallel
- ☐ Related data items that are used together are not organized into classes
- ☐ A routine uses more features of another class than of its own class
- ☐ A primitive data type is overloaded
- ☐ A class doesn't do very much
- ☐ A chain of routines passes tramp data
- ☐ A middle man object isn't doing anything
- ☐ One class is overly intimate with another
- ☐ A routine has a poor name
- ☐ Data members are public
- ☐ A subclass uses only a small percentage of its parents' routines
- ☐ Comments are used to explain difficult code
- ☐ Global variables are used
- ☐ A routine uses setup code before a routine call or takedown code after a routine call
- ☐ A program contains code that seems like it might be needed someday

Summary of Refactorings

Data Level Refactorings

- ☐ Replace a magic number with a named constant
- ☐ Rename a variable with a clearer or more informative name
- ☐ Move an expression inline
- ☐ Replace an expression with a routine
- ☐ Introduce an intermediate variable
- ☐ Convert a multi-use variable to a multiple single-use variables
- ☐ Use a local variable for local purposes rather than a parameter
- ☐ Convert a data primitive to a class
- ☐ Convert a set of type codes to a class
- ☐ Convert a set of type codes to a class with subclasses
- ☐ Change an array to an object
- ☐ Encapsulate a collection
- ☐ Replace a traditional record with a data class

Statement Level Refactorings

- ☐ Decompose a boolean expression
- ☐ Move a complex boolean expression into a well-named boolean function
- ☐ Consolidate fragments that are duplicated within different parts of a conditional
- ☐ Use break or return instead of a loop control variable
- ☐ Return as soon as you know the answer instead of assigning a return value within nested if-then-else statements
- ☐ Replace conditionals with polymorphism (especially repeated case statements)
- ☐ Create and use null objects instead of testing for null values

Routine Level Refactorings

- ☐ Extract a routine
- ☐ Move a routine's code inline
- ☐ Convert a long routine to a class
- ☐ Substitute a simple algorithm for a complex algorithm

- ☐ Add a parameter
- ☐ Remove a parameter
- ☐ Separate query operations from modification operations
- ☐ Combine similar routines by parameterizing them
- ☐ Separate routines whose behavior depends on parameters passed in
- ☐ Pass a whole object rather than specific fields
- ☐ Pass specific fields rather than a whole object
- ☐ Encapsulate downcasting

Class Implementation Refactorings

- ☐ Change value objects to reference objects
- ☐ Change reference objects to value objects
- ☐ Replace virtual routines with data initialization
- ☐ Change member routine or data placement
- ☐ Extract specialized code into a subclass
- ☐ Combine similar code into a superclass

Class Interface Refactorings

- ☐ Move a routine to another class
- ☐ Convert one class to two
- ☐ Eliminate a class
- ☐ Hide a delegate
- ☐ Replace inheritance with delegation
- ☐ Replace delegation with inheritance
- ☐ Remove a middle man
- ☐ Introduce a foreign routine
- ☐ Introduce a class extension
- ☐ Encapsulate an exposed member variable
- ☐ Remove Set() routines for fields that cannot be changed
- ☐ Hide routines that are not intended to be used outside the class
- ☐ Encapsulate unused routines
- ☐ Collapse a superclass and subclass if their implementations are very similar

System Level Refactorings

- ☐ Duplicate data you can't control
- ☐ Change unidirectional class association to bidirectional class association
- ☐ Change bidirectional class association to unidirectional class association
- ☐ Provide a factory routine rather than a simple constructor
- ☐ Replace error codes with exceptions or vice versa

Checklist: Refactoring Safely

- ☐ Is each change part of a systematic change strategy?
- ☐ Did you save the code you started with before beginning refactoring?
- ☐ Are you keeping each refactoring small?
- ☐ Are you doing refactorings one at a time?
- ☐ Have you made a list of steps you intend to take during your refactoring?
- ☐ Do you have a parking lot so that you can remember ideas that occur to you mid-refactoring?
- ☐ Have you retested after each refactoring?
- ☐ Have changes been reviewed if they are complicated or if they affect mission-critical code?
- ☐ Have you considered the riskiness of the specific refactoring, and adjusted your approach accordingly?
- ☐ Does the change enhance the program's internal quality rather than degrading it?
- ☐ Have you avoided using refactoring as a cover for code and fix or as an excuse for not rewriting bad code?

Chapter 25: Code-Tuning Strategies

Checklist: Code-Tuning Strategy

Overall Program Performance

- ☐ Have you considered improving performance by changing the program requirements?
- ☐ Have you considered improving performance by modifying the program's design?
- ☐ Have you considered improving performance by modifying the class design?
- ☐ Have you considered improving performance by avoiding operating system interactions?
- ☐ Have you considered improving performance by avoiding I/O?
- ☐ Have you considered improving performance by using a compiled language instead of an interpreted language?
- ☐ Have you considered improving performance by using compiler optimizations?
- ☐ Have you considered improving performance by switching to different hardware?
- ☐ Have you considered code tuning only as a last resort?

Code-Tuning Approach

- ☐ Is your program fully correct before you begin code tuning?
- ☐ Have you measured performance bottlenecks before beginning code tuning?
- ☐ Have you measured the effect of each code-tuning change?
- ☐ Have you backed out the code-tuning changes that didn't produce the intended improvement?
- ☐ Have you tried more than one change to improve performance of each bottleneck, i.e., iterated?

Chapter 26: Code Tuning Techniques

Checklist: Code-Tuning Techniques

Improve Both Speed and Size

- ☐ Substitute table lookups for complicated logic
- ☐ Jam loops
- ☐ Use integer instead of floating-point variables
- ☐ Initialize data at compile time
- ☐ Use constants of the correct type
- ☐ Precompute results
- ☐ Eliminate common subexpressions
- ☐ Translate key routines to assembler

Improve Speed Only

- ☐ Stop testing when you know the answer
- ☐ Order tests in case statements and if-then-else chains by frequency
- ☐ Compare performance of similar logic structures
- ☐ Use lazy evaluation
- ☐ Unswitch loops that contain if tests
- ☐ Unroll loops
- ☐ Minimize work performed inside loops
- ☐ Use sentinels in search loops
- ☐ Put the busiest loop on the inside of nested loops
- ☐ Reduce the strength of operations performed inside loops
- ☐ Change multiple-dimension arrays to a single dimension
- ☐ Minimize array references
- ☐ Augment data types with indexes
- ☐ Cache frequently used values
- ☐ Exploit algebraic identities
- ☐ Reduce strength in logical and mathematical expressions
- ☐ Be wary of system routines
- ☐ Rewrite routines in line

Chapter 28: Managing Construction

Checklist: Configuration Management

General

- ☐ Is your software-configuration-management plan designed to help programmers and minimize overhead?
- ☐ Does your SCM approach avoid overcontrolling the project?
- ☐ Do you group change requests, either through informal means such as a list of pending changes or through a more systematic approach such as a change-control board?
- ☐ Do you systematically estimate the effect of each proposed change?
- ☐ Do you view major changes as a warning that requirements development isn't yet complete?

Tools

- ☐ Do you use version-control software to facilitate configuration management?
- ☐ Do you use version-control software to reduce coordination problems of working in teams?

Backup

- ☐ Do you back up all project materials periodically?
- ☐ Are project backups transferred to off-site storage periodically?
- ☐ Are all materials backed up, including source code, documents, graphics, and important notes?
- ☐ Have you tested the backup-recovery procedure?

Chapter 29: Integration

Checklist: Integration

Integration Strategy

- ☐ Does the strategy identify the optimal order in which subsystems, classes, and routines should be integrated?
- ☐ Is the integration order coordinated with the construction order so that classes will be ready for integration at the right time?
- ☐ Does the strategy lead to easy diagnosis of defects?
- ☐ Does the strategy keep scaffolding to a minimum?
- ☐ Is the strategy better than other approaches?
- ☐ Have the interfaces between components been specified well? (Specifying interfaces isn't an integration task, but verifying that they have been specified well is.)

Daily Build and Smoke Test

- ☐ Is the project building frequently—ideally, daily to support incremental integration?
- ☐ Is a smoke test run with each build so that you know whether the build works?
- ☐ Have you automated the build and the smoke test?
- ☐ Do developers check in their code frequently—going no more than a day or two between check-ins?
- ☐ Is a broken build a rare occurrence?
- ☐ Do you build and smoke test the software even when you're under pressure?

Chapter 30: Programming Tools

Checklist: Programming Tools

- ☐ Do you have an effective IDE?
- ☐ Does your IDE support outline view of your program; jumping to definitions of classes, routines, and variables; source code formatting; brace matching or begin-end matching; multiple file string search and replace; convenient compilation; and integrated debugging?
- ☐ Do you have tools that automate common refactorings?
- ☐ Are you using version control to manage source code, content, requirements, designs, project plans, and other project artifacts?
- ☐ If you're working on a very large project, are you using a data dictionary or some other central repository that contains authoritative descriptions of each class used in the system?
- ☐ Have you considered code libraries as alternatives to writing custom code, where available?
- ☐ Are you making use of an interactive debugger?
- ☐ Do you use make or other dependency-control software to build programs efficiently and reliably?
- ☐ Does your test environment include an automated test framework, automated test generators, coverage monitors, system perturbers, diff tools, and defect tracking software?
- ☐ Have you created any custom tools that would help support your specific project's needs, especially tools that automate repetitive tasks?
- ☐ Overall, does your environment benefit from adequate tool support?

Chapter 31: Layout And Style

Checklist: Layout

General

- ☐ Is formatting done primarily to illuminate the logical structure of the code?
- ☐ Can the formatting scheme be used consistently?
- ☐ Does the formatting scheme result in code that's easy to maintain?
- ☐ Does the formatting scheme improve code readability?

Control Structures

- ☐ Does the code avoid doubly indented begin-end or {} pairs?
- ☐ Are sequential blocks separated from each other with blank lines?
- ☐ Are complicated expressions formatted for readability?
- ☐ Are single-statement blocks formatted consistently?
- ☐ Are case statements formatted in a way that's consistent with the formatting of other control structures?
- ☐ Have gotos been formatted in a way that makes their use obvious?

Individual Statements

- ☐ Is white space used to make logical expressions, array references, and routine arguments readable?
- ☐ Do incomplete statements end the line in a way that's obviously incorrect?
- ☐ Are continuation lines indented the standard indentation amount?
- ☐ Does each line contain at most one statement?
- ☐ Is each statement written without side effects?
- ☐ Is there at most one data declaration per line?

Comments

- ☐ Are the comments indented the same number of spaces as the code they comment?
- ☐ Is the commenting style easy to maintain?

Routines

- ☐ Are the arguments to each routine formatted so that each argument is easy to read, modify, and comment?
- ☐ Are blank lines used to separate parts of a routine?

Classes, Files and Programs

- ☐ Is there a one-to-one relationship between classes and files for most classes and files?
- ☐ If a file does contain multiple classes, are all the routines in each class grouped together and is the class clearly identified?
- ☐ Are routines within a file clearly separated with blank lines?
- ☐ In lieu of a stronger organizing principle, are all routines in alphabetical sequence?

Chapter 32: Self-Documenting Code

Checklist: Good Commenting Technique

General

- ☐ Can someone pick up the code and immediately start to understand it?
- ☐ Do comments explain the code's intent or summarize what the code does, rather than just repeating the code?
- ☐ Is the Pseudocode Programming Process used to reduce commenting time?
- ☐ Has tricky code been rewritten rather than commented?
- ☐ Are comments up to date?
- ☐ Are comments clear and correct?
- ☐ Does the commenting style allow comments to be easily modified?

Statements and Paragraphs

- ☐ Does the code avoid endline comments?
- ☐ Do comments focus on why rather than how?
- ☐ Do comments prepare the reader for the code to follow?
- ☐ Does every comment count? Have redundant, extraneous, and self-indulgent comments been removed or improved?
- ☐ Are surprises documented?
- ☐ Have abbreviations been avoided?
- ☐ Is the distinction between major and minor comments clear?
- ☐ Is code that works around an error or undocumented feature commented?

Data Declarations

- ☐ Are units on data declarations commented?
- ☐ Are the ranges of values on numeric data commented?
- ☐ Are coded meanings commented?
- ☐ Are limitations on input data commented?
- ☐ Are flags documented to the bit level?
- ☐ Has each global variable been commented where it is declared?
- ☐ Has each global variable been identified as such at each usage, by a naming convention, a comment, or both?
- ☐ Are magic numbers replaced with named constants or variables rather than just documented?

Control Structures

- ☐ Is each control statement commented?
- ☐ Are the ends of long or complex control structures commented or, when possible, simplified so that they don't need comments?

Routines

- ☐ Is the purpose of each routine commented?
- ☐ Are other facts about each routine given in comments, when relevant, including input and output data, interface assumptions, limitations, error corrections, global effects, and sources of algorithms?

Files, Classes, and Programs

- ☐ Does the program have a short document such as that described in the Book Paradigm that gives an overall view of how the program is organized?
- ☐ Is the purpose of each file described?
- ☐ Are the author's name, email address, and phone number in the listing?

Checklist: Self-Documenting Code

Classes

- ☐ Does the class's interface present a consistent abstraction?
- ☐ Is the class well named, and does its name describe its central purpose?
- ☐ Does the class's interface make obvious how you should use the class?
- ☐ Is the class's interface abstract enough that you don't have to think about how its services are implemented? Can you treat the class as a black box?

Routines

- ☐ Does each routine's name describe exactly what the routine does?
- ☐ Does each routine perform one well-defined task?
- ☐ Have all parts of each routine that would benefit from being put into their own routines been put into their own routines?
- ☐ Is each routine's interface obvious and clear?

Data Names

- ☐ Are type names descriptive enough to help document data declarations?
- ☐ Are variables named well?
- ☐ Are variables used only for the purpose for which they're named?
- ☐ Are loop counters given more informative names than i, j, and k?
- ☐ Are well-named enumerated types used instead of makeshift flags or boolean variables?
- ☐ Are named constants used instead of magic numbers or magic strings?
- ☐ Do naming conventions distinguish among type names, enumerated types, named constants, local variables, class variables, and global variables?

Data Organization

- ☐ Are extra variables used for clarity when needed?
- ☐ Are references to variables close together?
- ☐ Are data types simple so that they minimize complexity?
- ☐ Is complicated data accessed through abstract access routines (abstract data types)?

Control

- ☐ Is the nominal path through the code clear?
- ☐ Are related statements grouped together?
- ☐ Have relatively independent groups of statements been packaged into their own routines?
- ☐ Does the normal case follow the if rather than the else?
- ☐ Are control structures simple so that they minimize complexity?
- ☐ Does each loop perform one and only one function, as a well-defined routine would?
- ☐ Is nesting minimized?
- ☐ Have boolean expressions been simplified by using additional boolean variables, boolean functions, and decision tables?

Layout

- ☐ Does the program's layout show its logical structure?

Design

- ☐ Is the code straightforward, and does it avoid cleverness?
- ☐ Are implementation details hidden as much as possible?
- ☐ Is the program written in terms of the problem domain as much as possible rather than in terms of computer-science or programming-language structures?