Ideas to consider at different points in the Software Design and construction.

Based on https://github.com/xianshenglu/document/blob/master/Code%20Complete%202nd%20Edition.pdf

# **1) Design in Construction**

## **1.1) Design Heuristics**

* Find Real-World Objects
* Form Consistent Abstractions
* Encapsulate Implementation Details
* Inherit When possible
* Hide Secrets (Hide information)
* Identify Areas likely to change
* Keep Coupling Loose. (Independent Components as possible).
* Look for Common Design Patterns.

**Less Useful Heuristics**

* Aim for Strong Cohesion. (Components aim is clear).
* Build Hierarchies
* Formalize Class Contracts
* Assign Responsibilities
* Design for Test
* Avoid Failure
* Choose Binding Time Consciously
* Make Central Points of Control
* Consider Using Brute Force
* Draw a Diagram
* Keep Your Design Modular

## **1.2) Guidelines for Using Heuristics**

1. Understanding the Problem. You must understand the problem. What is the unknown? What’s the data? What is the condition? Is it possible to satisfy the condition? Is the condition enough to determine the unknown? Or is it insufficient? Or redundant? Or contradictory? Draw a figure. Introduce suitable notation. Separate the various parts of the condition. Can you write them down?

2. Devising a Plan. Find the connection between the data and the unknown. You might be obliged to consider auxiliary problems if you can't find an intermediate connection. You should eventually come up with a plan of the solution.

Have you seen the problem before? Or have you seen the same problem in a slightly different form? Do you know a related problem? Do you know a theorem that could be useful? Look at the unknown! And try to think of a familiar problem having the same or a similar unknown. Here is a problem related to yours and solved before. Can you use it? Can you use its result? Can you use its method? Should you introduce some auxiliary element to make its use possible? Can you restate the problem? Can you restate it still differently? Go back to definitions. If you cannot solve the proposed problem, try to solve some related problem first. Can you imagine a more accessible related problem? A more general problem? A more special problem? An analogous problem? Can you solve a part of the problem? Keep only a part of the condition, drop the other part; how far is the unknown then determined, how can it vary? Can you derive something useful from the data? Can you think of other data appropriate for determining the unknown? Can you change the unknown or the data, or both if necessary, so that the new unknown and the new data are nearer to each other? Did you use all the data? Did you use the whole condition? Have you considered all essential notions involved in the problem?

3. Carrying out the Plan. Carry out your plan. Carrying out your plan of the solution, check each step. Can you see clearly that the step is correct? Can you prove that it's correct?

4. Looking Back. Examine the solution. Can you check the result? Can you check the argument? Can you derive the result differently? Can you see it at a glance? Can you use the result, or the method, for some other problem?

## **1.3) 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 your 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, e-mail, flip charts, digital photography, 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 its parts strictly necessary?
* Do the design use standard techniques and avoid exotic, hard-to-understand elements?
* Overall, does the design help minimize both accidental and essential complexity?

## **1.4) Key Points**

* Software’s Primary Technical Imperative is managing complexity. This is greatly aided by a design focus on simplicity.
* Simplicity is achieved in two general ways: minimizing the amount of essential complexity that anyone’s brain must deal with at any one time and keeping accidental complexity from proliferating needlessly.
* Design is heuristic. Dogmatic adherence to any single methodology hurts creativity and hurts your programs.
* Good design is iterative; the more design possibilities you try, the better your final design will be.
* Information hiding is a particularly valuable concept. Asking “What should I hide?” settles many difficult design issues.

# **2) Class Design**

## **2.1) Summary of Reasons to Create a Class**

* Model real-world objects
* Model abstract objects
* Reduce complexity
* Isolate complexity
* Hide implementation details
* Limit effects of changes
* Hide global data
* Streamline parameter passing
* Make central points of control
* Facilitate reusable code
* Plan for a family of programs
* Package related operations
* Accomplish a specific refactoring

## **2.2) 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—that is, do derived classes adhere to the Liskov Substitution Principle?
* Does the class documentation describe the inheritance strategy?
* Do derived classes avoid “overriding” non-overridable routines?
* Are common interfaces, data, and behaviour as high as possible in the inheritance tree?
* Are inheritance trees 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 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?

## **2.3) Key Points**

* Class interfaces should provide a consistent abstraction. Many problems arise from violating this single principle.
* A class interface should hide something—a system interface, a design decision, or an implementation detail.
* Containment is usually preferable to inheritance unless you’re modelling an “is a” relationship.
* Inheritance is a useful tool, but it adds complexity, which is counter to Software’s Primary Technical Imperative of managing complexity.
* Classes are your primary tool for managing complexity. Give their design as much attention as needed to accomplish that objective.

# **3) Routines/Functions/Methods**

## **3.1) Summary list of the valid reasons for creating a routine:**

* Reduce complexity
* Introduce an intermediate, understandable abstraction
* Avoid duplicate code
* Support sub classing
* Hide sequences
* Hide pointer operations
* Improve portability
* Simplify complicated Boolean tests
* Improve performance

**In addition, many of the reasons to create a class are also good reasons to create a routine:**

* Isolate complexity
* Hide implementation details
* Limit effects of changes
* Hide global data
* Make central points of control
* Facilitate reusable code
* Accomplish a specific refactoring

## **3.2) CHECKLIST: High-Quality Routines**

**Big-Picture Issues**

* Is the reason for creating the routine enough?
* 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, 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?

## **3.3) Key Points**

* The most important reason for creating a routine is to improve the intellectual manageability of a program, and you can create a routine for many other good reasons. Saving space is a minor reason; improved readability, reliability, and modifiability are better reasons.
* Sometimes the operation that most benefits from being put into a routine of its own is a simple one.
* You can classify routines into various kinds of cohesion, but you can make most routines functionally cohesive, which is best.
* The name of a routine is an indication of its quality. If the name is bad and it’s accurate, the routine might be poorly designed. If the name is bad and it’s inaccurate, it’s not telling you what the program does. Either way, a bad name means that the program needs to be changed.
* Functions should be used only when the primary purpose of the function is to return the specific value described by the function’s name.
* Careful programmers use macro routines with care and only as a last resort.

# **4) Defensive Programming**

## **4.1) CHECKLIST: Defensive Programming**

**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 favour robustness or correctness?
* Have barricades been created to contain the damaging effect of errors and reduce the amount of code that must be concerned about error processing?
* Have debugging aids been used in the code?
* 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 nonlocal 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?

## **4.2) Key Points**

* Production code should handle errors in a more sophisticated way than “garbage in, garbage out.”
* Defensive-programming techniques make errors easier to find, easier to fix, and less damaging to production code.
* Assertions can help detect errors early, especially in large systems, high-reliability systems, and fast-changing code bases.
* The decision about how to handle bad inputs is a key error-handling decision and a key high-level design decision.
* Exceptions provide a means of handling errors that operates in a different dimension from the normal flow of the code. They are a valuable addition to the programmer’s intellectual toolbox when used with care, and they should be weighed against other error-processing techniques.
* Constraints that apply to the production system do not necessarily apply to the development version. You can use that to your advantage, adding code to the development version that helps to flush out errors quickly.

# **5) Pseudocode Programming Process**

## **5.1) 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 mainly 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 by 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?

## **5.2) Key Points**

* Constructing classes and constructing routines tends to be an iterative process.
* Insights gained while constructing specific routines tend to ripple back through the class’s design.
* Writing good pseudocode calls for using understandable English, avoiding features specific to a single programming language and writing at the level of intent (describing what the design does rather than how it will do it).
* The Pseudocode Programming Process is a useful tool for detailed design and makes coding easy. Pseudocode translates directly into comments, ensuring that the comments are accurate and useful.
* Don’t settle for the first design you think of. Iterate through multiple approaches in pseudocode and pick the best approach before you begin writing code.
* Check your work at each step and encourage others to check it too. That way, you’ll catch mistakes at the least expensive level, when you’ve invested the least amount of effort.

# **6) General Considerations in Using Data**

## **6.1) 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? (And have you turned on all the available warnings?)
* 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 reference 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, are you 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?

## **6.2) Key Points**

* Data initialization is prone to errors, so use the initialization techniques described in this chapter to avoid the problems caused by unexpected initial values.
* Minimize the scope of each variable. Keep references to a variable close together.
* Keep it local to a routine or class. Avoid global data.
* Keep statements that work with the same variables as close together as possible.
* Early binding tends to limit flexibility but minimize complexity. Late binding tends to increase flexibility but at the price of increased complexity.
* Use each variable for one and only one purpose.

# **7) Naming Variables**

## **7.1) 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 misread or 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?

## **7.2) Key Points**

* Good variable names are a key element of program readability. Specific kinds of variables such as loop indexes and status variables require specific considerations.
* Names should be as specific as possible. Names that are vague enough or general enough to be used for more than one purpose are usually bad names.
* Naming conventions distinguish among local, class, and global data. They distinguish among type names, named constants, enumerated types, and variables.
* Regardless of the kind of project you’re working on, you should adopt a variable naming convention. The kind of convention you adopt depends on the size of your program and the number of people working on it.
* Abbreviations are rarely needed with modern programming languages. If you do use abbreviations, keep track of abbreviations in a project dictionary or use the standardized prefixes approach.
* Code is read far more times than it is written. Be sure that the names you choose favour read-time convenience over write-time convenience.

# **8) Fundamental Data**

## **8.1) 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 name constants for data declarations and loop limits rather than magic numbers?
* Have named constants been used consistently—not used as named constants in some places and as 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?

## **8.2) Key Points**

* Working with specific data types means remembering many individual rules for each type. Use this chapter’s checklist to make sure that you’ve considered the common problems.
* Creating your own types makes your programs easier to modify and more self documenting, if your language supports that capability.
* When you create a simple type using typedef or its equivalent, consider whether you should be creating a new class instead.