Everything You Know is Wrong

A Guide to Real-Time Programming

by Sheldon Linker

Over the years, many advances have been made in computer science. For every advance, we have gained a rule-of-thumb to guide us to this advance. We have also had a number of untested, wild-assed theories foisted upon us, also with their rules of thumb. These rules-of-thumb are just that — not hard-and-fast rules, but general guidelines. Some of these rules are always valid. Some are valid under some circumstances. Others are never valid. Below, I list some rules, and explain my categorization of them.

1. **The user manual guides everything.**
   The user manual is the first and the last; the only requirements document that really means anything. Whatever the user manual says must be adhered to. Nobody will ever know if you ignored requirement X103b(7). However, if you violate the sentence describing what the blue button does, we'll all know right away.

2. **The Rapid Development Model: A Slow and Painful Death**
   The Rapid Development Model is only rapid if you don't know where you're going because nothing of the sort has been invented yet. There is no escaping the fact that using this model of development, things will be scrapped and started over. In any system that is supposed to fulfill a known function, such as doing what the user manual says, the best approach is to completely design the entire system, down to the interfaces between components at least.

3. **Separate the project into components.**
   A good idea, but what does a component mean? Separating a project down to a number of separate, separable components, typically talking to each other through some sort of messaging scheme is great when your system must be spread over a number of processors. However, when the system is to be placed into a single processor, every message-passing interface is both costly in terms of design time, coding time, testing time, execution time, and code space. Instead, remember that components may be just conceptual things. First, separate the project into processes, absolutely minimizing these processes. The minimum number of processes is the number of tasks that must occur at the same time. If the program must do A and B at the same time, then A and B belong in different processes. If the program does B, then C, then B and C belong in the same process.

4. **Computers are fast, and memory is big.**
   Flat-out wrong. If you're doing real-time programming (this includes any programming involving a user waiting for any sort of a reply), then the computer is <u>not</u> fast enough. If you have any limits on memory at all, then act like there's almost no memory.

5. **System integration is the step between unit test and delivery.**
   Wrong again. System build and integration is an integral part of the design and programming process. I have actually been on a project where the last step was to connect the two boxes, and one had a square 100-pin male connector, and the other had a round 64-pin male connector.

6. **There will be time to fix that later.**
   No. You will be in a time crunch later. Fix it now. Remember, the first half of the work takes 80% of the time. And so does the second half.

7. The OS is extensible.

Sure, the OS is extensible, but never extend it for any reason unless you must. This means that if there is any other way to do what you want, then that item doesn't belong in the OS. Unix is a good example of this. All of the level (3) routines are not part of the OS. They're in the API layer. If it can go in an API layer, it should.

8. Code is reusable. Write your code so that it can be reused in another project.

Unless you are writing your code as a part of a general API, it won't be reused. Don't waste your time pretending that it will be. On the other hand, spend a few minutes at the start of the project. Are there items that should be part of a company-wide API? Something that's not already out there, but reflects how your company does business, or some look-and-feel particular to your company? If so, make it an API, and publish its description internally.

9. Code is reusable, part 2.

Project-wide, there should be a list of storage items (data outside of classes), routines (again outside of classes), structures (structure definitions), constant sets (single items and enumerations), and of course, classes. Before you write a routine, check to see if the item you need is on the list as already specified (and if you're lucky, coded and maybe even tested).

10. Code is reusable, part 3.

Remember that there are libraries of code outside of your area. Before you specify a routine or class to be written internally, check to see if its already out there, as a part of the standard library or class structure, or as shareware or 3rd party software.

11. Information-hiding

Information hiding, you have been told, makes code more reusable, and leads to fewer errors. It prevents programmers from relying on things they shouldn't rely on. True, but there are better ways to do this. There are several levels of information hiding. Let's take the example of a variable that's made private, such as temperature. Very often, the concept of temperature is still directly exposed, but using getTempurature and setTemperature. You have hidden nothing. You've just used a more complicated way to expose the data. Information hiding can also work directly against you. Let's say that temperature is stored as a floating-point number, in Kelvin, and that in order to fully hide the information, you expose getTempKFloat, getTempKInt, getTempCFloat, getTempCInt, getTempFFloat, and getTempFInt. There is nothing here that tells the caller that using getTempKFloat is the fastest. Bad programmers benefit from information hiding, because they're likely to get in less trouble. Good programmers benefit from all items being public, because they can use the information to better the overall code. What about refactoring? Don't be afraid to revise code. With all data public, a change to a lower level routine may cause refactoring at a higher level. With most data private, the higher level code is never in an optimized state, and will never be refactored to keep it optimized.

12. Getters and Setters

Getters and setters are the perfect means to make sure that anyone doesn't do anything untoward to a variable, and that your code can be spread across a multitude of processors without you having to notice. Only in a real-time system, your code will all be on the same processor, and you don't have time or space for getters or setters, except when you're getting something that isn't a field, or setting a field has side-effects. If setting a field has side effects, then you must have a set routine. This doesn't mean that the field must be made private. In polite society, the field can be documented and commented as "Set only with Set routine".

Bad example:

```
private int x;
private int twiceX;

int getX() {
     return x;
}

void setX(int newX) {
     x = newX;
     twiceX = 2 * x;
}

void actualSample() {
     setX(getX() + 1);
}
```

Good example:

```
public  int x; // *** SET USING ROUTINE ONLY ***
private int twiceX;

void setX(int newX) {
     x = newX;
     twiceX = 2 * x;
}

void actualSample() {
     setX(x + 1);
}
```

13. Write each component in the best language.

This should really be "Write each process in the best language", or perhaps "Write the entire system in the best language". Where you run into problems is at the language interface. First, let me explain that interfacing some languages is not a problem. For instance, in interfacing similar languages, like C, C++, and Objective C, you can use the same H files and pass the same binary structures from one item to another, so for all practical purposes, the C language family is a single language with multiple dialects. A second level of language hops would be using something like PL/1 and Cobol. PL/1 and Cobol store their structures differently, but in PL/1, you can declare an interface or structure to be of the Cobol type, so again, no problem in making the leap. However, at this level of language interface, we need to duplicate our header files. Example:

C H file:

```
#define myConst 1
```

PL/1 include file:

```
%myConst = 1;
```

The next level of language-hopping would be one in which the call stacks differ. For instance, on many systems, C calls with the arguments on the stack, where Fortran calls with a pointer to the argument block on the stack. The argument block has the number of arguments and their addresses. Here, extra code in the C routine must be used to read the argument block. At this point, calling from one language to another becomes a pain, but still doable, and with very little timing overhead.

At the next level of language difference, we have, for instance, calling C from Java. At this point, we need a JNI conversion, which involves a lot of code, and relatively more time to execute. No data structures can be directly shared between the two languages, and each read or write of anything requires the use of a large and time-consuming getter or setter. The lack of shared data structures can cause immense amounts of code to be required. At this level we have the additional problem that no debugger in the world can make the transition across the JNI line.

At the highest level of language difference, we have two programs, written as separate tasks trying to communicate with each other. Again, this type of communication means that data structures can't be shared in their simplest binary form, since Java doesn't speak binary.

In summary, stay in the same language family throughout the entire project, unless there is a serious and overriding reason not to.

14. Java is as fast as compiled code, given a good VM.

Java runs on the Java Virtual Machine. Since there is no machine manufactured with the same structure as the Java VM, I can reasonably guarantee you that the structure in use is not fully efficient on your machine. A translate-as-you-go VM will quickly reach the fastest possible efficiency for the machine structure it is emulating, but it will never reach the speed of your machine. For instance,

```
x = a * b + c * d;
```

will be coded as

```
PSH A
PSH B
MUL
PSH C
PSH D
MUL
ADD
POP X
```

This may get translated to assembly, modifying items on the JVM stack directly. However, it will never get translated to

```
; Variable X stored in R1
MOV R1,A
MUL R1,B
MOV R3,C
MUL R3,D
ADD R1,R3
```

15. Separating things into functional layers leads to better code.

Separating things out into functional layers leads to routines which are easier to understand than the routines would have been had they been coded to traverse multiple conceptual layers. However, what we get is more routines, so that the overall system may be harder to understand, because there are more routines. Doing separation for separation's sake alone may well lead to more effort in code writing, a larger code base (which is harder to understand overall), and slower operation. If there is a technical reason to separate things into layers, then by all means do so. But, if the separation is purely because you're supposed to, then skip the effort.

16. Overloading names

The common wisdom is that using the same name over and over again for methods is a good idea. For instance, you have class A and class B, and they both do a Save operation, so you write a Save method for both. Another example is that you have class C, with subclasses CA and CB. Since C has a Save method, subclasses CA and CB may override it with their own Save method. In theory, this is great. However, when it comes to the real world, this can cause an immense amount of learning time for debuggers or those new to the team. Why? Because when you look for a Save routine, you'll find hundreds of them. What's the alternative? Simple. A gets ASave, B gets BSave, C gets CSave, CA gets CASave, and CB gets CBSave. There are two important exceptions to this, because there are two times when you must use the same name. One is in the case when you are overriding a method that will already get called by something. The other exception is exemplified in the C, CA, and CB case. If you will be instantiating CA and CB objects, but passing them somewhere as objects of-or-derived-from class C, then the Save method in all 3 classes must have the same name.

17. Overloading operators

If you do this, at all, ever, you're insane. Your code will be completely self-obsfuscating to the point where even you will not know what your code is doing. There is one exception to this rule: You may override an operator to throw an error, stating that the operator doesn't make sense in some particular setting.

18. Each level of subroutine should make the minimal conceptual change.

Again, this leads to small routines, each of which is very easy to understand. However, it also leads to a large number of routines, making the overall system hard to understand. For instance, if I want to take the hypotenuse of 3 items, I can do it inline, on one line, or I can have a Hypotenuse3 routine call a Hypotenuse routine call a Square and Sqrt routine. There are two rules here:

(1) Some things are too trivial to make separate routines.

(2) If a routine has only 1 caller, then it shouldn't be a routine.

There's a corollary to this:

(2a) If a variable or field is only set once and only read once (or read once for each set, directly following, and without logic in-between), then the expression which the variable or field is set to should be used in place of the variable or field, and the item eliminated.

19. Tight code doesn't matter.

If your computer has infinite speed and infinite storage, then tight code doesn't matter. The fact that faster computers are coming out later is immaterial. Your program is going out now. Also, the infinitely fast machine with infinite storage isn't coming out next year or the year after. Also, remember that smaller code is generally faster code. Smaller code is often cheaper to write than is verbose code. Smaller code is easier to understand, too.

20. It's best to be on-time.

Very often, there is a trade-off between being on-time and correct. The fallacy here is that you can turn in a project that doesn't work quite right, on time. You can't. The reason is that the customer or QA will look at the incomplete project and declare it not done. Then it's not on time. Actually, it's not on time if QA catches it. You're dead meat if the customer catches it.

21. Object-oriented code is best.

It used to be the case that people wrote completely procedural code, with no objects. Some jobs are obviously object-oriented. For instance, performing a set of actions, such as open, close, resize, print, save, et cetera, on a variety of window types. Subclass the application window, and make a set of methods that do all of these things. Other tasks, however, are not so object-oriented. Some tasks are purely procedural. For instance, let's say that we have to do some processing on a number of entries in a file. If the line type is X, we will do processing X, then Y. If type Y, then we'll do processing Y then X. For any other type, we'll do processing Z. Obviously, we'll need mainline code, and we'll need subroutines X and Y. If we're using a language in which the use of objects is optional, such as C++, then in this instance there is no reason whatsoever to make any classes or objects. We can do the job procedurally without incurring any object overhead.

22. Avoid static objects.

The advocates of object-oriented programming tell us that having static objects is a thing of the past; that everything should be encapsulated in objects. The object oriented proponents have a term for classes containing exactly one object: Singletons. What purpose do singleton objects serve? They serve to place data that belongs in static storage into an object. In doing so, they cost extra code, which costs extra money while writing the program, and extra time when executing it. Here's an example in Java:

Singleton method:

```
class MySingleton {
   private static MySingleton mySingleton = null;
   private int calledTimes;

   MySingleton() {
      calledTimes = 0;
   }

   static synchronized MySingleton GetInstance() {
      if (mySingleton==null)
         mySingleton = new MySingleton();
   }

   void ActiveRoutine() {
      System.out.println("I have been called " +
         ++calledTimes + " times");
   }
}

MySingleton.GetInstance.ActiveRoutine();
```

Static method:

```
class MySingleton {
   private static int calledTimes = 0;

   static void ActiveRoutine() {
      System.out.println("I have been called " +
         ++calledTimes + " times");
   }
}

MySingleton.ActiveRoutine();
```

In C, you can leave the class out altogether in this type of code:

```
void MySingletonActiveRoutine(void) {
   static int calledTimes;

   printf("I have been called %d times\n", ++calledTimes);
}

MySingletonActiveRoutine();
```

23. Warnings

If you're writing code that will go into production, then there should be no warnings whatsoever that apply to your code. What do I mean by "apply"? Simple: Some classes of warnings always apply to your code, such as invariant conditional expressions. Other classes or warnings may or may not apply to a project. For instance, in a project which won't ever be ported to another platform, you can turn off warnings about portability. All that said, all available remaining classes of warnings should be treated as errors. Why? Because every single one of them is either (a) something that is actually an error, or (b) something that will cause you or some other programmer at a later date to investigate whether the warning is an error. In each case, determine whether the compiler's warning points out a real error, a spot in which your code can be simplified, or is a false alarm. Obviously, you must fix the real errors, and should fix the areas for improvement. For each false alarm, fix the code in such a way that the compiler is happy, but without causing extra compiled code. Here is an example:

Before:

```
long a = f();
short b = a;
      Warning:  Significant digits may be lost by assignment
if (x=y)
   z();
      Warning:  Possible unintended setting of x
for (i=0; i<100 && c[i]; i++)
   ;
      Warning:  Loop body is empty
```

After:

```
long a = f();
short b = (short)a;
x = y;
if (x)
   z();
for (i=0; i<100 && c[i]; i++)
   /*EMPTY*/;
```

24. Abstraction is good.

Abstraction seems to have two meanings in object-oriented coding.

Abstraction, as it pertains to classes which are never instantiated, but instead are subclassed and only then instantiated are a necessary part of the definition of a class tree.

Abstraction, as it pertains to layer-upon-layer of subroutine sets is a waste of time, except when you are trying to match an existing, external interface. For instance, let's say that you've got some new OS that's to be Posix-compliant. Of course, you would write a layer of subroutines which abstract the native OS services to look like the Posix definition. However, if you're writing internal code, you don't need extra layers of abstraction. It's usually better to have the high-end code use the low-end code, with knowledge of what's going on, rather than writing a middle-abstraction layer. Remember, every piece of knowledge or design you hide will be used in a less-than-optimal way.

25. XML, RPC, Soap, and the like are good ways to exchange data.

If your data must go through some completely generic interface, then perhaps. However, if your message is going from one of your programs to another one of your programs, then the message should be tailored into a format such that the minimum code should be needed to write the message, and the minimum code should be needed to read the message. In commercial applications, this means that a minimum number of lines of code should be written. In real-time applications, this means that a minimum number of instructions should be executed, even if that means writing more lines of code.

If a message is being sent from or to a Java application, or other type of application that doesn't deal with packed binary data, then text is OK. However, numbers are better than strings for execution speed. For instance, let's say that I wanted to transmit a message setting one or more parameters to certain values. There are a variety of ways to do this, shown from most time-consuming to least. In the examples, I will show sample messages to set Horizontal to 1 and Vertical to 2.

The XML way:

Definitions (C version):

```
#define COORDINATE_MESSAGE "Coordinates"
#define HORIZONTAL "Horizontal"
#define VERTICAL "Vertical"
#define  DEPTH "Depth"
```

Sample ASCII or UTF-8 message:

```
<Coordinates>
     <Horizontal>1</Horizontal>
     <Vertical>2</Vertical>
</Coordinates>
```

The application-specific way, involving Java-like applications:

Definitions (C version)

```
#define COORDINATE_MESSAGE 1
enum {HORIZONTAL, VERTICAL, DEPTH};
```

Sample ASCII message:

```
1 0 1 1 2
```

The application-specific way, involving native applications:

Definitions in C are the same.

Sample binary message (shown in hex):

01000001010002

26. Named constants are better than raw constants.

Usually, yes. However, there are some constants which are just not worth naming. For instance, let's say that you have some field that can have the values 480, 720, or 1080, and that there are no real names for these values. Don't make constants named FV480, FV720, and FV1080. Just use the numbers.