Software Development Best Practices: Part 2 OOP

-- quotes from Steve McConnell, Code Complete 2nd Ed

If you can't figure out how to use a class based solely on its interface documentation, the right response is not to pull up the source code and look at the implementation. That's good initiative but bad judgment. The right response is to contact the author of the class and say "I can't figure out how to use this class.

Watch for coupling that's too tight. "Coupling" refers to how tight the connection is between two classes. In general, the looser the connection, the better. Several general guidelines flow from this concept: Minimize accessibility of classes and members. Avoid friend classes, because they're tightly coupled. Make data private rather than protected in a base class to make derived classes less tightly coupled to the base class. Avoid exposing member data in a class's public interface. Be wary of semantic violations of encapsulation. Observe the "Law of Demeter" (discussed in Design and Implementation Issues of this chapter). Coupling goes hand in glove with abstraction and encapsulation. Tight coupling occurs when an abstraction is leaky, or when encapsulation is broken.

Containment is the simple idea that a class contains a primitive data element or object. A lot more is written about inheritance than about containment, but that's because inheritance is more tricky and error-prone, not because it's better. Containment is the work-horse technique in object-oriented programming.

Be critical of classes that contain more than about seven data members.

Inheritance is the idea that one class is a specialization of another class. The purpose of inheritance is to create simpler code by defining a base class that specifies common elements of two or more derived classes. The common elements can be routine interfaces, implementations, data members, or data types. Inheritance helps avoid the need to repeat code and data in multiple locations by centralizing it within a base class. When you decide to use inheritance, you have to make several decisions: For each member routine, will the routine be visible to derived classes? Will it have a default implementation? Will the default implementation be overridable? For each data member (including variables, named constants, enumerations, and so on), will the data member be visible to derived classes?

Implement "is a" through public inheritance. When a programmer decides to create a new class by inheriting from an existing class, that programmer is saying that the new class "is a" more specialized version of the older class. The base class sets expectations about how the derived class will operate and imposes constraints on how the derived class can operate (Meyers 1998).

If the derived class isn't going to adhere completely to the same interface contract defined by the base class, inheritance is not the right implementation technique. Consider containment or making a change further up the inheritance hierarchy.

Inheritance adds complexity to a program, and, as such, it's a dangerous technique. As Java guru Joshua Bloch says, "Design and document for inheritance, or prohibit it." If a class isn't designed to be inherited from, make its members non-virtual in C++, final in Java, or non-overridable in Microsoft Visual Basic so that you can't inherit from it.

Here's a summary of when to use inheritance and when to use containment: If multiple classes share common data but not behavior, create a common object that those classes can contain. If multiple classes share common behavior but not data, derive them from a common base class that defines the common routines. If multiple classes share common data and behavior, inherit from a common base class that defines the common data and routines. Inherit when you want the base class to control your interface; contain when you want to control your interface.

Because it's a poor tradeoff to add complexity for dubious performance gains, a good approach to deep vs. shallow copies is to prefer deep copies until proven otherwise.

Isolate complexity. Complexity in all forms, complicated algorithms, large data sets, intricate communications protocols, and so on, is prone to errors. **If an error does occur, it will be easier to find if it isn't spread through the code but is localized within a class**. Changes arising from fixing the error won't affect other code because only one class will have to be fixed, other code won't be touched. If you find a better, simpler, or more reliable algorithm, it will be easier to replace the old algorithm if it has been isolated into a class. During development, it will be easier to try several designs and keep the one that works best.
Hide global data. If you need to use global data, you can hide its implementation details behind a class interface. Working with global data through access routines provides several benefits compared to working with global data directly. You can change the structure of the data without changing your program. You can monitor accesses to the data. The discipline of using access routines also encourages you to think about whether the data is really global; it often becomes apparent that the "global data" is really just object data.
Streamline parameter passing. If you're passing a parameter among several routines, that might indicate a need to factor those routines into a class that share the parameter as object data. Streamlining parameter passing isn't a goal, per se, but passing lots of data around suggests that a different class organization might work better.
NASA's Software Engineering Laboratory studied ten projects that pursued reuse aggressively (McGarry, Waligora, and McDermott 1989). In both the object-oriented and the functionally oriented approaches, the initial projects weren't able to take much of their code from previous projects because previous projects hadn't established a sufficient code base. Subsequently, the projects that used functional design were able to take about 35 percent of their code from previous projects. Projects that used an object-oriented approach were able to take more than 70 percent of their code from previous projects. If you can avoid writing 70 percent of your code by planning ahead, do it!
NASA identifies reuse candidates at the ends of their projects. They then perform the work needed to make the classes reusable as a special project at the end of the main project or as the first step in a new project. This approach helps prevent "gold-plating" the creation of functionality that isn't required and that unnecessarily adds complexity.
Here's a summary list of the valid 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…
Reduce complexity. The single most important reason to create a routine is to reduce a program's complexity. Create a routine to hide information so that you won't need to think about it.
One indication that a routine needs to be broken out of another routine is deep nesting of an inner loop or a conditional. Reduce the containing routine's complexity by pulling the nested part out and putting it into its own routine.
Avoid duplicate code. Undoubtedly the most popular reason for creating a routine is to avoid duplicate code. Indeed, creation of similar code in two routines implies an error in decomposition. Pull the duplicate code from both routines, put a generic version of the common code into a base class, and then move the two specialized routines into subclasses.
Correctness means never returning an inaccurate result; returning no result is better than returning an inaccurate result. Robustness means always trying to do something that will allow the software to keep operating, even if that leads to results that are inaccurate sometimes. Safety-critical applications tend to favor correctness to robustness. It is better to return no result than to return a wrong result. The radiation machine is a good example of this principle. Consumer applications tend to favor robustness to correctness. Any result whatsoever is usually better than the software shutting down.

A common programmer blind spot is the assumption that limitations of the production software apply to the development version. The production version has to run fast. The development version might be able to run slow. The production version has to be stingy with resources. The development version might be allowed to use resources extravagantly. The production version shouldn't expose dangerous operations to the user. The development version can have extra operations that you can use without a safety net.
One of the paradoxes of defensive programming is that during development, you'd like an error to be noticeable able, you'd rather have it be obnoxious than risk overlooking it. But during production, you'd rather have the error be as unobtrusive as possible, to have the program recover or fail gracefully.
During production, your users need a chance to save their work before the program crashes and they are probably willing to tolerate a few anomalies in exchange for keeping the program going long enough for them to do that. Users don't appreciate anything that results in the loss of their work, regardless of how much it helps debugging and ultimately improves the quality of the program. If your program contains debugging code that could cause a loss of data, take it out of the production version.
Hacking? Some programmers try to hack their way toward working code rather than using a systematic approach like the pseudocode programming process (PPP). If you've ever found that you've coded yourself into a corner in a routine and have to start over, that's an indication that the PPP might work better. If you find yourself losing your train of thought in the middle of coding a routine, that's another indication that the PPP would be beneficial. Have you ever simply forgotten to write part of a class or part of routine? That hardly ever happens if you're using the PPP. If you find yourself staring at the computer screen not knowing where to start, that's a surefire sign that the PPP would make your programming life easier.