Principle-Driven Development

by Chuck Allison

GETTY IMAGES

If it hasn't happened to you yet, it will. No matter how experienced you are, you will run up against a programming challenge that will overwhelm you. As Frederick Brooks states in "No Silver Bullet: Essence and Accidents of Software Engineering":

> *Einstein argued that there must be simplified explanations of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer. Much of the complexity that he must master is arbitrary complexity . . . The complexity of software is an essential property, not an accidental one.*

A colleague of mine recently described a complex PBX system he worked on at Bell Labs. It consisted of almost seven million lines of C and C++ code. Since the success of a strategic business communication system depended on that software, high reliability and availability were absolutely essential. Strict real-time constraints and system integrity had to be maintained dynamically; otherwise the system would be unusable.

How do you successfully navigate all this unavoidable complexity? With so much change and evolution you might ask, "What is permanent? What is dependable? What can we take from one project, environment, or methodology to another?"

The answer is *principles*—basic laws or truths on which we can depend. Sometimes they are captured in catchy little phrases that a novice may not fully appreciate. How many times have you heard the following pithy yet practical proverbs?

- "Strive for loosely coupled abstractions that interact."
- "Don't repeat yourself."
- "Say it in code."
- "Test early. Test often."

Platitudinous, to be sure, but they are guidelines based on proven principles.

Consider "Don't repeat yourself," a saying made popular by the Pragmatic Programmers. C developers have always known that it is much better to say `#define MAXLINE 81` than it is to scatter the numeric literal 81 throughout their code. What's the underlying principle here? "Every entity should have only one definition." I just made that up—maybe you can say it better.

Once you've decided that it is not a good idea to write repetitive code, it can still sneak up on you. In learning C++, for example, students eventually reach a point where they have to understand how classes such as `string` are actually implemented. Here's a first go:

```
class String
{
   char* data;
public:
   String(const char* str = "") {
      data = new char[strlen(str) + 1];
      strcpy(data,str);
   }
   ~String() {
      delete [] data;
   }
};
```

Later, when they learn about copy constructors and assignment operators, they might add code like this:

```
   String(const String& s) {
      data = new char[strlen(s.data) + 1];
      strcpy(data, s.data);
   }
   String& operator=(const String& s) {
      if (this != &s) {
         char* newData = new char[strlen(s.data) + 1];
         strcpy(newData, s.data);
         delete [] data;
         data = newData;
      }
      return *this;
   }
```

Now they have a `String` class that behaves like a built-in type, complete with deep-copy semantics. But they also have repeated code. Hopefully they see it and will refactor it into a separate function, like `dup` below, and call it as needed.

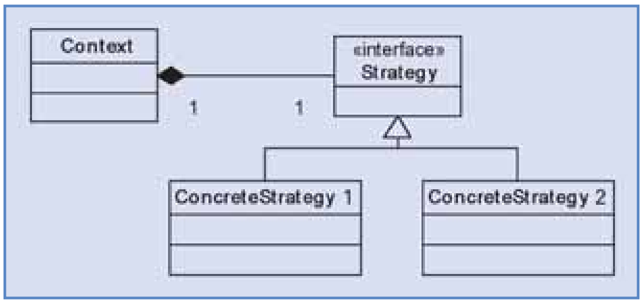
```
   char* dup(const char* from) {  // Common code
      char* to = new char[strlen(from) + 1];
      return strcpy(to, from);
  }
```

Principles and Patterns

A basic tenet of object-oriented design dictates that clients of a component should not depend on the details of its implementation. Instead, they should use a component's services through a well-defined interface. The introduction to the Gang of Four book, *Design Patterns*, states:

> *There are two benefits to manipulating objects solely in terms of the interface defined by abstract classes:*
>
> 1. *Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.*
> 2. *Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class(es) defining the interface.*
>
> *This so greatly reduces implementation dependencies between subsystems that it leads to the following principle of reusable object-oriented design:*
>
> *Program to an interface, not an implementation.*

Every design pattern in existence adheres to this important principle. Strategy, for instance, suggests grouping related implementations under a common interface, giving users a polymorphic handle to a concrete implementation through an interface reference. The canonical class sketch illustrates the idea like this:



But it is not necessary to always have an explicit interface for the family of implementations. The standard C++ container adaptors, `queue`, `stack`, and `priority_queue`, use low-level sequences like vector to store their data. The definition of `queue` looks like this:

```
template <class T, class Container = deque<T> >
class queue { … };
```

Its storage mechanism, which defaults to `deque`, is a template parameter. You could use `vector`, `list`, or any container of your own making as long as it provides the expected functionality. In other words, the expected interface is *implicit*. Encapsulating what varies (the underlying implementation) and separating it from what stays the same (the `queue` interface) adheres to the principle of programming to an interface instead of an implementation and, in this case, is a compile-time realization of the principles behind Strategy.

Small Beginnings

One of my favorite principles is at the very core of effective software engineering. I first saw it in writing in Grady Booch's *Object-Oriented Analysis and Design*, where he quoted the following from John Gall:

> *A complex system that works is invariably found to have evolved from a simple system that worked . . . A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a simple system.*

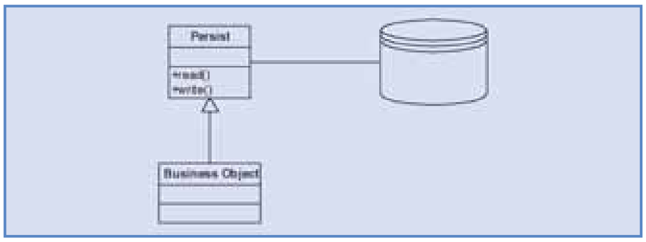
I believe this very principle spawned the incremental development theme so widely seen in software projects these days and that it applies to design as well as to implementation.

In the 1980s, I was part of a team developing a shop-floor control system at a large defense contractor. A fleet of PC workstations with touchscreens was to be networked to a UNIX data server. Quality control specialists on the floor would visually inspect circuit boards for defects and touch the corresponding area on a graphic on their screens with a stylus. My first task was to write the software that ran each workstation. Waterfall methodologies were all the rage and object-oriented design was no more than a gleam in Grady Booch's eye, yet we somehow knew to start small and to grow the system in an incremental fashion. Here's what my first pass at the workstation executive looked like:

```
for (;;)
{
    user_login();
    build_menu();
    do_option();
    free_tasks();
}
```

Putting that little bit of stubbed code into a main program that compiled and executed had an enabling effect on me. I had begun with a very simple system. The intervening months to project completion saw many iterations adding bar code-reading functionality, logging, etc., but everything grew from that simple beginning.

With apologies to Yogi Berra, it was "déjà vu all over again" a decade later. Client-server infrastructure was more fantasy than fact then, and I was in charge of developing a "reusable" persistence layer in a three-tier environment for business objects whose data was stored in corporate relational databases. "Reuse" was the buzzword du jour, but few people actually knew how to achieve it, including us (or at least we thought we didn't know how). Naturally we hired consultants—three rounds of them—but we didn't get much closer to a solution. Once again, I fell back on something simple. Here's the UML diagram for what I had in mind:



Not a stroke of brilliance, but I tweaked it, made working code out of it, and once again gained momentum from a very simple system. After some months of two-week iterations—during which we solved the object-hierarchy-to-table-mapping problem, resolved connectivity issues, and developed a tool to produce ready-to-use C++ code from database tables—a quality reusable C++ framework emerged.

This principle now has an entrenched reification in the agile world: Walking Skeleton. The idea is to make the rudiments of a system's architecture tangible early in a project's lifecycle. From there you simultaneously build up infrastructure and functionality (see the StickyNotes for more information).

The take-away from all of this is that while formal methodologies and practices have their place, we shouldn't let them obscure the underlying principles that govern the creation of quality software. If we don't take them too seriously, accepted practices and methodologies can lead us to a principle-centered state of mind where simple, creative solutions come more naturally. **{end}**

Chuck Allison developed software for twenty years before becoming a professor of computer science at Utah Valley State College. He was senior editor of the C/C++ Users Journal *and is founding editor of* The C++ Source. *He is also the author of two C++ books and gives onsite training in C++, Python, and design patterns.*

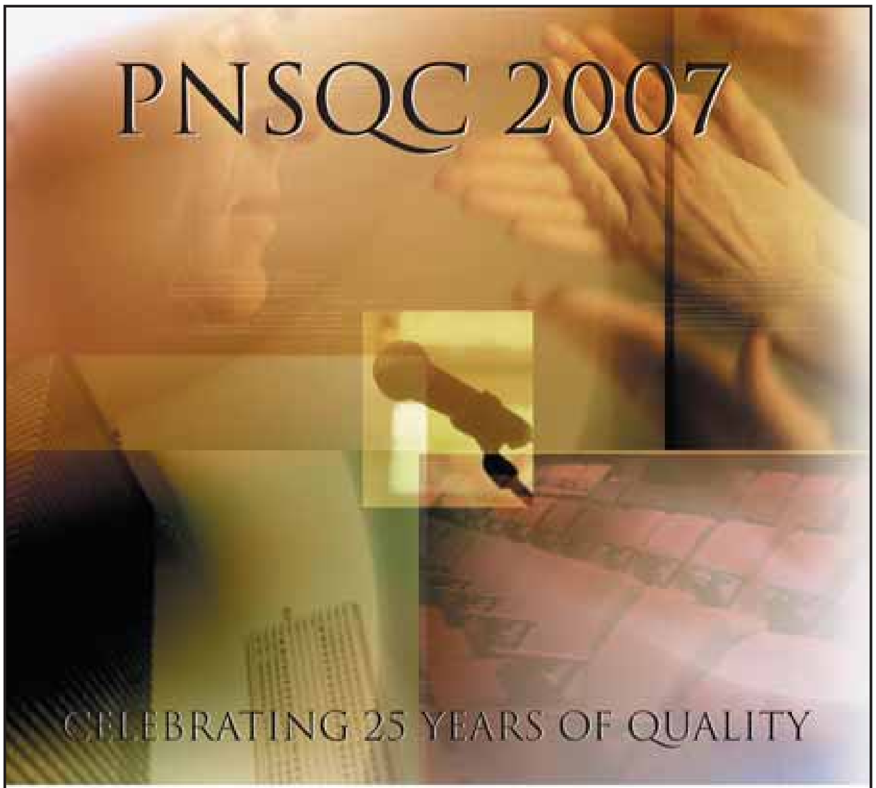
Sticky Notes

For more on the following topics, go to www.StickyMinds.com/bettersoftware.

- References
- Walking Skeleton

There are many principles that guide effective software development—only a few have been mentioned here. Which principles have guided you to success?

Follow the link on the StickyMinds.com homepage to join the conversation.



ANNOUNCING THE
2007 CALL FOR ABSTRACTS

BUILDING FOR A BETTER FUTURE

SHARE IDEAS AND GAIN RECOGNITION

On October 8 – 10, 2007, at the Oregon Convention Center, PNSQC will celebrate 25 years of bringing together a community of software professionals keenly interested in software quality.

To be an integral part of our celebration we invite you to submit one or more abstracts of papers related to quality, especially those that speak to our conference theme – **Building for a Better Future**.

If your topic is selected, you receive complimentary admission to the two-day technical program portion of the conference where you present the resulting paper.

TO SUBMIT AN ABSTRACT OR FIND OUT MORE
visit www.pnsqc.org

25TH ANNUAL
PACIFIC NW
SOFTWARE QUALITY CONFERENCE