The Evils of Duplication

Giving a computer two contradictory pieces of knowledge was Captain James T. Kirk's preferred way of disabling a marauding artificial intelligence. Unfortunately, the same principle can be effective in bringing down your code.

As programmers, we collect, organize, maintain, and harness knowledge. We document knowledge in specifications, we make it come alive in running code, and we use it to provide the checks needed during testing.

Unfortunately, knowledge isn't stable. It changes—often rapidly. Your understanding of a requirement may change following a meeting with the client. The government changes a regulation and some business logic gets outdated. Tests may show that the chosen algorithm won't work. All this instability means that we spend a large part of our time in maintenance mode, reorganizing and reexpressing the knowledge in our systems.

Most people assume that maintenance begins when an application is released, that maintenance means fixing bugs and enhancing features. We think these people are wrong. Programmers are constantly in maintenance mode. Our understanding changes day by day. New requirements arrive as we're designing or coding. Perhaps the environment changes. Whatever the reason, maintenance is not a discrete activity, but a routine part of the entire development process.
When we perform maintenance, we have to find and change the representations of things—those capsules of knowledge embedded in the application. The problem is that it’s easy to duplicate knowledge in the specifications, processes, and programs that we develop, and when we do so, we invite a maintenance nightmare—one that starts well before the application ships.

We feel that the only way to develop software reliably, and to make our developments easier to understand and maintain, is to follow what we call the DRY principle:

**EVERY PIECE OF KNOWLEDGE MUST HAVE A SINGLE, UNAMBIGUOUS, AUTHORITATIVE REPRESENTATION WITHIN A SYSTEM.**

Why do we call it DRY?

The alternative is to have the same thing expressed in two or more places. If you change one, you have to remember to change the others, or, like the alien computers, your program will be brought to its knees by a contradiction. It isn’t a question of whether you’ll remember: it’s a question of when you’ll forget.

You’ll find the DRY principle popping up time and time again throughout this book, often in contexts that have nothing to do with coding. We feel that it is one of the most important tools in the Pragmatic Programmer’s tool box.

In this section we’ll outline the problems of duplication and suggest general strategies for dealing with it.

**How Does Duplication Arise?**

Most of the duplication we see falls into one of the following categories:

- **Imposed duplication.** Developers feel they have no choice—the environment seems to require duplication.

- **Inadvertent duplication.** Developers don’t realize that they are duplicating information.
- **Impatient duplication.** Developers get lazy and duplicate because it seems easier.

- **Interdeveloper duplication.** Multiple people on a team (or on different teams) duplicate a piece of information.

Let's look at these four i's of duplication in more detail.

**Imposed Duplication**

Sometimes, duplication seems to be forced on us. Project standards may require documents that contain duplicated information, or documents that duplicate information in the code. Multiple target platforms each require their own programming languages, libraries, and development environments, which makes us duplicate shared definitions and procedures. Programming languages themselves require certain structures that duplicate information. We have all worked in situations where we felt powerless to avoid duplication. And yet often there are ways of keeping each piece of knowledge in one place, honoring the DRY principle, and making our lives easier at the same time. Here are some techniques:

**Multiple representations of information.** At the coding level, we often need to have the same information represented in different forms. Maybe we're writing a client-server application, using different languages on the client and server, and need to represent some shared structure on both. Perhaps we need a class whose attributes mirror the schema of a database table. Maybe you're writing a book and want to include excerpts of programs that you also will compile and test.

With a bit of ingenuity you can normally remove the need for duplication. Often the answer is to write a simple filter or code generator. Structures in multiple languages can be built from a common metadata representation using a simple code generator each time the software is built (an example of this is shown in Figure 3.4, page 106). Class definitions can be generated automatically from the online database schema, or from the metadata used to build the schema in the first place. The code extracts in this book are inserted by a preprocessor each time we format the text. The trick is to make the process active: this cannot be a one-time conversion, or we're back in a position of duplicating data.
**Documentation in code.** Programmers are taught to comment their code: good code has lots of comments. Unfortunately, they are never taught *why* code needs comments: bad code *requires* lots of comments.

The DRY principle tells us to keep the low-level knowledge in the code, where it belongs, and reserve the comments for other, high-level explanations. Otherwise, we're duplicating knowledge, and every change means changing both the code and the comments. The comments will inevitably become out of date, and untrustworthy comments are worse than no comments at all. (See *It's All Writing*, page 248, for more information on comments.)

**Documentation and code.** You write documentation, then you write code. Something changes, and you amend the documentation and update the code. The documentation and code both contain representations of the same knowledge. And we all know that in the heat of the moment, with deadlines looming and important clients clamoring, we tend to defer the updating of documentation.

Dave once worked on an international telex switch. Quite understandably, the client demanded an exhaustive test specification and required that the software pass all tests on each delivery. To ensure that the tests accurately reflected the specification, the team generated them programmatically from the document itself. When the client amended their specification, the test suite changed automatically. Once the team convinced the client that the procedure was sound, generating acceptance tests typically took only a few seconds.

**Language issues.** Many languages impose considerable duplication in the source. Often this comes about when the language separates a module's interface from its implementation. C and C++ have header files that duplicate the names and type information of exported variables, functions, and (for C++) classes. Object Pascal even duplicates this information in the same file. If you are using remote procedure calls or CORBA [URL 29], you'll duplicate interface information between the interface specification and the code that implements it.

There is no easy technique for overcoming the requirements of a language. While some development environments hide the need for header files by generating them automatically, and Object Pascal allows you to abbreviate repeated function declarations, you are generally stuck with
what you’re given. At least with most language-based issues, a header file that disagrees with the implementation will generate some form of compilation or linkage error. You can still get things wrong, but at least you’ll be told about it fairly early on.

Think also about comments in header and implementation files. There is absolutely no point in duplicating a function or class header comment between the two files. Use the header files to document header interface issues, and the implementation files to document the nitty-gritty details that users of your code don’t need to know.

Inadvertent Duplication

Sometimes, duplication comes about as the result of mistakes in the design.

Let’s look at an example from the distribution industry. Say our analysis reveals that, among other attributes, a truck has a type, a license number, and a driver. Similarly, a delivery route is a combination of a route, a truck, and a driver. We code up some classes based on this understanding.

But what happens when Sally calls in sick and we have to change drivers? Both Truck and DeliveryRoute contain a driver. Which one do we change? Clearly this duplication is bad. Normalize it according to the underlying business model—does a truck really have a driver as part of its underlying attribute set? Does a route? Or maybe there needs to be a third object that knits together a driver, a truck, and a route. Whatever the eventual solution, avoid this kind of unnormalized data.

There is a slightly less obvious kind of unnormalized data that occurs when we have multiple data elements that are mutually dependent. Let’s look at a class representing a line:

```java
class Line {
    public:
        Point start;
        Point end;
        double length;
};
```

At first sight, this class might appear reasonable. A line clearly has a start and end, and will always have a length (even if it’s zero). But we
have duplication. The length is defined by the start and end points: change one of the points and the length changes. It's better to make the length a calculated field:

```java
class Line {
    public:
        Point start;
        Point end;
        double length() { return start.distanceTo(end); }
}
```

Later on in the development process, you may choose to violate the DRY principle for performance reasons. Frequently this occurs when you need to cache data to avoid repeating expensive operations. The trick is to localize the impact. The violation is not exposed to the outside world: only the methods within the class have to worry about keeping things straight.

```java
class Line {
    private:
        bool changed;
        double length;
        Point start;
        Point end;
    public:
        void setStart(Point p) { start = p; changed = true; }
        void setEnd(Point p) { end = p; changed = true; }
        Point getStart(void) { return start; }
        Point getEnd(void) { return end; }
        double getLength() {
            if (changed) {
                length = start.distanceTo(end);
                changed = false;
            }
            return length;
        }
}
```

This example also illustrates an important issue for object-oriented languages such as Java and C++. Where possible, always use accessor functions to read and write the attributes of objects.¹ It will make it easier to add functionality, such as caching, in the future.

---

¹ The use of accessor functions ties in with Meyer's Uniform Access principle [Mey97b], which states that "All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation."
Impatient Duplication

Every project has time pressures—forces that can drive the best of us to take shortcuts. Need a routine similar to one you've written? You'll be tempted to copy the original and make a few changes. Need a value to represent the maximum number of points? If I change the header file, the whole project will get rebuilt. Maybe I should just use a literal number here; and here; and here. Need a class like one in the Java runtime? The source is available, so why not just copy it and make the changes you need (license provisions notwithstanding)?

If you feel this temptation, remember the hackneyed aphorism “short cuts make for long delays.” You may well save some seconds now, but at the potential loss of hours later. Think about the issues surrounding the Y2K fiasco. Many were caused by the laziness of developers not parameterizing the size of date fields or implementing centralized libraries of date services.

Impatient duplication is an easy form to detect and handle, but it takes discipline and a willingness to spend time up front to save pain later.

Interdeveloper Duplication

On the other hand, perhaps the hardest type of duplication to detect and handle occurs between different developers on a project. Entire sets of functionality may be inadvertently duplicated, and that duplication could go undetected for years, leading to maintenance problems. We heard firsthand of a U.S. state whose governmental computer systems were surveyed for Y2K compliance. The audit turned up more than 10,000 programs, each containing its own version of Social Security number validation.

At a high level, deal with the problem by having a clear design, a strong technical project leader (see page 228 in Pragmatic Teams), and a well-understood division of responsibilities within the design. However, at the module level, the problem is more insidious. Commonly needed functionality or data that doesn't fall into an obvious area of responsibility can get implemented many times over.

We feel that the best way to deal with this is to encourage active and frequent communication between developers. Set up forums to discuss common problems. (On past projects, we have set up private Usenet
newsgroups to allow developers to exchange ideas and ask questions. This provides a nonintrusive way of communicating—even across multiple sites—while retaining a permanent history of everything said.) Appoint a team member as the project librarian, whose job is to facilitate the exchange of knowledge. Have a central place in the source tree where utility routines and scripts can be deposited. And make a point of reading other people’s source code and documentation, either informally or during code reviews. You’re not snooping—you’re learning from them. And remember, the access is reciprocal—don’t get twisted about other people poring (pawing?) through your code, either.

Tip 12
Make It Easy to Reuse

What you’re trying to do is foster an environment where it’s easier to find and reuse existing stuff than to write it yourself. If it isn’t easy, people won’t do it. And if you fail to reuse, you risk duplicating knowledge.

Related sections include:
- Orthogonality, page 34
- Text Manipulation, page 99
- Code Generators, page 102
- Refactoring, page 184
- Pragmatic Teams, page 224
- Ubiquitous Automation, page 230
- It’s All Writing, page 248