# What Makes Product Code Different

January 30, 2017

Todd Morley

Alteryx Analytic Products

## Introduction

This is not a coding-standards document. It’s a thought paper that outlines a few principles underlying many coding standards. A young software developer who adheres to these principles can write pretty good product code even without the guidance of coding standards. To the degree that effective coding standards require agreement about conventions, a team of developers can apply the same principles to define appropriate language-specific standards. This paper’s aim is thus to equip a young development team to elaborate well-founded coding standards, in spite of the team’s inexperience.

## Total Cost of Ownership

1. **Software development is product development.**

When software is a product, the firm derives benefit from the software to the extent that the software’s value in the marketplace exceeds the software’s total cost of ownership (TCO) to the firm. So a software-development team should minimize the firm’s TCO, for a given set of features.

1. **Most of the cost of software development lies in maintaining code, not writing it.**

Coding practices that make code more maintainable pay for themselves. Writing modular code is a good example, because modular code separates interface from implementation, making it easy to change implementation without breaking dependent code. Re-using code, and writing re-usable code, is another example, because it avoids the necessity of maintaining several copies of the same code (resulting from copy-paste coding).

Coding practices that avoid code-maintenance activities have high value. Code maintenance usually means fixing bugs and other forms of technical debt. Automated testing discovers bugs before QA or release, avoiding the need for post-release maintenance. So while test-driven development means writing a lot more (test) code, it more than pays for itself in avoided maintenance.

1. **Most of the cost of maintaining product code lies in comprehending legacy code and testing changes to it, not writing those changes.**

Code should be self-documenting, in the sense that an entry-level programmer should be able to discern the code’s structure and function in a single reading.

* Self-documenting code uses complete and exact terms to name variables, functions, arguments, etc.
* Self-documenting code uses prefixes, suffixes, and similar decorations to identify objects’ types and roles.
* Self-documenting code includes comments above every non-obvious function call or operation, explaining the non-obvious code in enough detail to make its structure and function obvious.
* Self-documenting code uses indentation and grouping to emphasize and clarify hierarchical structure (making the code narrow and deep rather than wide and shallow).
* Self-documenting code implements object-oriented and relational coding idioms consistently. For example, self-documenting code implements create, read, update, and delete (CRUD) operations using similarly named and defined functions, on top of all database tables.
* Self-documenting code has the same style across the code base, regardless of how many developers have contributed to the code base. The code base reads as if a single developer authored the entire code base.
* Self-documenting code is to software development what plain prose is to writing. It makes clear thinking look easy, not clever.

1. **The cost of changing a code module is very superlinear in the time lag between writing the module and changing it.**

Good coding practices move bug discovery (or avoidance) as far upstream as possible. Idiomatic coding practices such as enforcing specific types of defensive-coding and exception-handling idioms avoids bugs before they’re ever coded. Test automation catches most bugs before code checkin. Bugs that escape testing at a given stage of the development process at time t become tests at that stage (at worst) at time t + 1. Test-coverage metrics and case analyses drive test development.

## User Experience

1. **The value a firm derives from software products (net of the software’s total cost of ownership) derives from user experience.**

While engineering does not directly influence sales performance, we influence it indirectly, by creating products that sell themselves. Good coding practices increase velocity and quality enough to enable us to deliver a rich user experience by shifting development activities from maintenance to new-feature development.

1. **User experience reduces to correctness, performance, and ease of use.**

Clever but opaque code does not contribute *per se* to these goals. If an entry-level programmer will have trouble deciphering the code at a single reading, we must weigh the difficulty of code maintenance against the possible benefit to the three user-experience criteria. We only use clever coding techniques when they make a substantial difference in these areas, or when the clever techniques substantially reduce TCO. And in such cases we write ample code comments that make the clever technique transparent.

1. **Consistency over time and across different parts of the user interface is a key driver of positive user experience.**

Idiomatic, re-usable code drives consistency of user experience. We therefore use the same functions repeatedly for the same tasks, uniformly across the entire code base, writing re-usable code whenever we reasonably anticipate the possibility of re-use. When we consider coding a new function, we ask ourselves whether any other module already implements the functionality we contemplate. If so, we refactor the pre-existing code as necessary to re-use it, rather than writing a second version of it from scratch.

1. **The user interface should be sufficiently idiomatic and self-explanatory for users to use it without first receiving training or reading manuals. (This criterion defines self-service software.)**

An idiomatic UI results naturally from idiomatic source code. A self-explanatory UI results from an idiomatic UI whose idioms are well conceived and easily identifiable, and which contains ample context-specific online documentation at all levels of granularity (field, form, page, module). Online documentation thus becomes part of the code base, and is developed with an eye to the same readability norms as we apply to the code base proper.