Essential Programming Paradigm

Claude Y. Knaus

claude.knaus@gmail.com

Abstract

The chronic difficulty of software maintenance can be traced back to widely held assumptions that inhibit progress in computer science. In the last instance, the idiosyncrasies of programming paradigms must be held accountable for bad software design. The characteristics of an alternative programming paradigm without such drawbacks is outlined.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General

General Terms Design, Languages

Keywords Design Patterns, Domain Specific Languages, Programming Paradigm

Hard Software

In software engineering, the maintenance of software is more expensive than its development. When a large software product matures, even apparently simple feature additions or bug fixes become obstacles. Frederick Brooks writes in *The Mythical Man-Month* (1): *Lehman and Belady have studied the history of successive releases in a large operating system. They find that the total number of modules increases linearly with release number, but that the number of modules affected [by changes introduced during the release] increases exponentially.* As the tasks become larger and harder to estimate, software projects become more expensive and less predictable. Ultimately, the progression of maintenance costs hits the worst case scenario: rewrite or retirement of the entire product.

Nothing seems to have changed since Brooks' monumental book, released over 30 years ago. When a problem persists for so long, strongly held assumptions must be questioned.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.
ACM 978-1-60558-220-7/08/10.

Myth #1: General Purpose Programming Languages

Software maintenance requires a software product to remain flexible despite its continuous growth. To ensure this flexibility is the purpose of software design. By anticipating patterns of change, the software product is structured in a way such that common and related changes have local impact and independent changes happen in distinct locations. Without software design, even simple change requests can have global impact.

The outcome of software design depends on the usual suspects: process, project management, methodology, people, and tools. While most of these factors can be changed with relative ease, programming languages stand out as an exception: they are irrevocably decided upon, early in the process of software development. Once a program is cast in a particular language, it is nearly impossible to translate it into another one. Lacking alternatives, a software product is tied to a programming language for good.

However, the consequences of choosing one programming language over another are not well understood. It does not help either that this choice is often arbitrarily made; when the future cannot be anticipated, familiarity of developers with a programming language is given precedence over other criteria.

Due to this tight coupling and uncertainty, a crucial requirement of a programming language is its generality. Many programming languages are labeled as *general purpose*, implying that they are applicable to almost any software project. Knowing that many programming languages are competing for this label, one has to ask how general they really are. Obviously, the common denominator of all programming languages is Turing-completeness. It is the one property which gives the software engineer the confidence and comfort that any computational problem can be solved, at least in principle. This property by itself may entitle a programming language as general.

Unfortunately, there is no agreement on generality beyond Turing-completeness. While programming languages have a functional requirement that they can tackle any problem, they also have an aesthetic aspect with respect to *how* they solve problems. Intuitively, this aspect cannot be objectively judged. What is considered a general programming

language has changed over the past from machine languages, Fortran, Cobol, to the languages we use today. Problems have pushed the evolution of programming languages, which in turn exposed new problems; the tools and problems that are solved by them co-evolve and define each other. In practice, no existing programming language can be considered a general problem solver.

Myth #2: Design Patterns are Good

Good software design is often mentioned in the same breath with design patterns. There is, however, a common misconception that design patterns themselves stand for good design.

Common practices like agile programming and test-driven development promote rigorous refactoring between steps of functional implementations. The idea of refactoring is that common code patterns distributed in the software are isolated into a single location in order to simplify future changes. After the refactoring process, the software product is considered in the state of "good software design".

However, there are certain common code patterns that withstand refactoring: design patterns. From the perspective of refactoring, design patterns stand for everything but good software design. They are repetitive code patterns which are scattered throughout the project, connected by invisible and implicit dependencies. They obscure the essential code by interleaving them with a bulk of seemingly unrelated code. Yet, limited by constraints of the programming language, they defy refactoring.

The reason why design patterns are still considered good is due to the fact that any project is tied to a programming language. Without the alternative to change the language, the perception of design patterns changes dramatically. After a refactoring process, code patterns that cannot be factored emerge as design patterns. Indeed, the authors of *Design Patterns* admit that most design patterns were found *after* refactoring (2). The identification of design patterns signals the developer that there is nothing more to refactor, no matter how the code may seem to disagree. Design patterns present a clear demarcation for refactoring, which is why they can be considered indicators for the rest of the software to be well designed.

What is lost in the ongoing euphoria for design patterns is the critique of the programming languages for failing to provide common features. The software engineers, stuck with the choice of the programming language, cannot be blamed; they welcome the workaround.

Domain-Paradigm Mismatch

Design patterns are not specific to the programming language, but to the underlying paradigm. The original design patterns were found for Smalltalk and C++, both considered representatives of the OO-paradigm. Other programming paradigms provoke different design patterns, even for generally "common problems". The functional paradigm introduces monads as design patterns for the lack of states; the imperative paradigm has adopted the entire OO-paradigm as a design pattern; logic programming uses design patterns for efficient control flow and avoidance of infinite recursion. Thanks to Turing-completeness, when commonly used programming features are missing, they can be implemented in terms of other paradigm constructs. However, since these features are still missing at the paradigm level, they cannot be reused by reference, but must be duplicated every time the feature is required.

Programming languages and IDEs attempt to ease the maintenance of design patterns through usability features like syntactic sugar, macros, templates, code assist, quick fix and automatic refactoring. As they only fight the symptoms of design pattern proliferation, they are no substitute for missing paradigm features. In desperate situations, large pattern-heavy projects resort to meta-programming and code generation, with the result that language specific features like debugging and profiling are compromised.

Design patterns not only depend on the programming paradigm, but also on the applied problem domain. This may not be obvious as most design patterns like decorator or singleton address common programming problems. However, design patterns emerge whenever a programming paradigm does not match the domain.

Domains are defined by typical problems which are being solved in the given domain. Ideally, if every domain concept is given respect by the paradigm, a domain expert can express the problems effectively and will never encounter design patterns.

Traditional programming paradigms, although disguised by their host languages as "general purpose", cannot fulfill such a requirement for all domains. This is because existing programming paradigms are biased; They promote favored concepts, like objects, functions, or rules, while neglecting others. By declaring their love for a particular concept, they define a class of problems, which they are tuned to solve. In other words, all existing programming paradigms have a bias toward domains where the paradigm concepts dominate. Hybrid- and multi-paradigm languages obviously cover larger domains, but not all.

Domain Specific Languages?

Domain specific languages (DSL) attempt to close the gap by shaping the paradigm to match the domain. DSLs are created through programming paradigms known as model-driven architecture or language-oriented programming. The basic idea is that design patterns are avoided by pushing them into a higher meta-level. The result of the integration into the meta-program converts the design pattern into a paradigm feature of the DSL.

For a DSL to be successful, the problem domain must be sufficiently mature. Unfortunately, real world domains

are rarely stable let alone well defined. Domains are constantly in flux; when problems have been solved, new problems arise which expand the horizon of the domain. As new problems are being encountered and solved, the gap between the domain and the paradigm widens again and new design patterns emerge. A domain expert is dependent on the meta-programmer to integrate the latest design patterns.

Unfortunately, meta-programming is not immune to design patterns either. Depending on the source and target domains they connect, design patterns arise in meta-programs as well. The logical step is to eliminate these design patterns by stepping up another meta-level. Although programs tend to become smaller with increasing meta-level and thus the recursion is expected to be limited, multiple meta-levels pose obstacles to understanding and debugging.

All in all, DSLs are heavyweight constructs which do not increase flexibility, but add new boundaries by distinguishing meta-levels. Meta-programming can only move but not eliminate design patterns. Also, there is little point in explicitly matching a domain which lacks sharp definition. Due to the dynamics of real-world domains, the success of a DSL to clear design patterns can only be temporary.

Essence of Programming

To understand how to avoid design patterns, we have to look at their structure. A design pattern consists of two parts, a part which is common to all instances of the pattern, and a part which is specific to the specific instantiation. The two parts are interleaved in a way such that they cannot be separated by any means of the programming paradigm. There are two reasons responsible for such structures:

1. *Limited composition*. What differentiates design patterns from other types of repetitive code that can be refactored lies in parameterization. For example, a function allows dynamic customization of its general behavior through arguments acting as parameters. This can be considered run-time composition of dynamic arguments with static code. However, not all paradigm concepts are allowed for parameterization. Typically, values and references are accepted arguments, but not types or statements. This limitation has led to the popularity of templates, which extend parameterization to types.
2. *Clustered features*. Solving a domain problem through composition may require paradigm features which are bundled with other features. The use of one feature implies dragging along others, needed or not. Examples in Java would be functor objects where a function drags along an object or the "misuse" of arrays and objects for functions returning multiple values. The inseparability of paradigm features gives design patterns their specific structural character. While the semantic baggage can be optimized away by the compiler, the syntactical baggage remains.

Summarizing, structures like design patterns occur because the programming paradigm disallows certain run-time composition of dynamic and static code. Faced with this limitation, the developer is forced to perform this composition at development-time by creating many static combinations, eventually leading to design pattern proliferation. Templates, if available, then represent compromises by positioning the composition task halfway at compile-time. The aesthetic aspect of the pattern is then further impacted by excess elements required to implement the composition.

It follows from the problem of limited composition that, to avoid design patterns, we have to allow any kind of composition at run-time of the code. This can only work if every possible programming composition can also be performed at run-time. In other words, the programming paradigm must be dynamic.

However, the composition is not an arbitrary one. It has to exhibit Turing-completeness, for otherwise, nothing could be implemented. To counter the problem of clustered features, the concepts forming the composition shall be clearly separate and elementary.

This would then be a bias free paradigm. A paradigm that consists of a composition of components which exhibit Turing-completeness has no central, favored concept other than the property of Turing-completeness itself. This would be a paradigm just about computation.

Conclusions

The limiting factor of software maintenance has been identified as today's programming paradigms. Design patterns emerge as symptoms of mismatch between domains and paradigms, making software less maintainable. All existing programming paradigms suffer from this problem due to their bias toward some domain, inevitably excluding other domains. The presumed essential programming paradigm is entirely bias free and would be the first paradigm deserving to be labeled general purpose.

References

[1] Brooks, Frederick, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Anniversary ed. Addison-Wesley, 1995.

[2] Gamma, Erich, Richard Helm, Ralph Johnson, & John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.