Programming languages, frameworks, and platforms require the developer to use a collection of provided programming features—abstractions—to express data, implement desired calculations, interact with other technologies, create user interfaces, and so on. A collection of coherent, often ideologically or theoretically based abstractions constitutes a *programming paradigm*. Often, a given programming technology is based on one particular paradigm.

Well-known examples include object-oriented, relational, functional, constraint-based, theorem-proving, concurrent, imperative, and declarative. Less well-known (or perhaps less well-defined) examples include graphical, reflective, context-aware, rule-based, and agent-oriented.

A particular paradigm leads to a specific type of implementation style and is best suited to certain types of applications. Relational programming benefits information-rich applications, whereas imperative programming is commonly used to control hardware. But today’s applications are seldom homogeneous. They are frequently complex systems, made up of many subcomponents that require a mixture of technologies. Thus, using just one language technology and paradigm is becoming much less common, replaced by *multiparadigm programming* in which the heterogeneous application consists of several subcomponents, each implemented with an appropriate paradigm and able to communicate with other subcomponents imple-
mented with a different paradigm. When more than one language is used, we call this polyglot (“many tongues”) programming.

Internet and intranet applications are inherently heterogeneous, often combining languages like JavaScript, Flash, and HTML in the user interface (UI), Java, C#, and Ruby in the middle tier, and SQL in the database. Not only are many languages involved, but there are also many paradigms, with SQL implementing the relational model, and object-oriented programming (OOP) dominating the middle and UI tiers. Even some “single-tier” desktop and server applications mix implementation languages, such as a fast, compiled “kernel” language for performance-sensitive components and a scripting language for integrating those components to create features, make extensibility easier, and so on.

Combining paradigms offers important benefits—for example, OOP minimizes the conceptual gap between the problem domain and the implementation in software, and functional programming (FP) brings mathematical rigor and robustness to computing, especially for concurrent applications. Historically, most of the world’s data has been managed via the relational model, although alternative persistence approaches are becoming more common. Applications for some domains are best organized using forms of logic programming, such as constraint-based solvers. However, using more than one language and paradigm increases the number of tools, libraries, and best practices that the development team must manage. As always, there are costs and benefits in these decisions.

This special issue explores the resurgent trend of multiparadigm programming and the related trend of polyglot programming. We call this a resurgent trend because it isn’t a new practice, but it has seemed to wane for many years. It became common for developers to focus on one language, such as C++ or Java, and one paradigm, usually OOP. Many developers thought of themselves as “Java programmers,” for example, emphasizing their single-language focus. Similarly, most acquired little knowledge of paradigms other than OOP. A probable source for this trend was the explosive growth of the Internet in the mid 1990s. A few environments, such as Java and .NET, dominated industry attention. Because OOP had become very popular in the early 1990s, and OOP was the dominant paradigm in most of the popular languages, developers learned and used OOP exclusively.

The Case for Multiparadigm Programming

Recent trends, such as the proliferation of concurrency and distribution as essential scalability tools, virtual machines, and new requirements for general-purpose, reusable platforms, have revived interest in multiparadigm and polyglot programming.

FP is arguably the oldest paradigm in software development, yet it was largely an academic concern until recently. Mainstream adoption has grown in response to the growing use of concurrency to scale applications. More developers must be able to write robust concurrent code, yet the traditional mainstay of OO and procedural systems, lock-based concurrency (synchronized access to shared, mutable state), is very difficult to master. The properties of FP, such as immutable values and side-effect-free functions, are the foundation for robust concurrency models, such as actors (http://en.wikipedia.org/wiki/Actor_model), which are also easier to learn and use. We’re finding that FP improves code quality in many other ways as well.

Erlang (www.erlang.org) is the best example of the successful use of FP and actors in industrial use. Created at Ericsson, it has been used to implement very robust telecom switches with thousands of concurrent processes. That success has spurred adoption by other projects requiring robust scalability, such as Riak (http://wiki.basho.com/display/RIAK/Riak), CouchDB (http://couchdb.apache.org), and GitHub (http://github.com). Actor libraries now exist in many other languages.

On the other hand, the benefits of OOP include encapsulation mechanisms and intuitive ways to model complex domains in software. OOP is a natural fit for GUIs, which probably drove the mainstream adoption of OOP in the 1980s, when GUIs also went mainstream. Once prevalent, OOP also proved broadly applicable.

In some ways, FP and OOP appear at odds. (The same can be said for combinations involving other paradigms listed previously.) For example, FP’s immutable values and side-effect-free functions seem at odds with mutable object state, a hallmark of OOP. However, we can exploit the best features of both paradigms if we restrict objects to be immutable or use persistent data structures and managed references (http://en.wikipedia.org/wiki/Persistent_data_structure) for controlled mutability (as in Clojure, http://clojure.org/data_structures), and we introduce side effects safely into FP using monads (http://en.wikipedia.org/wiki/Monad_(functional_programming)).
Which Languages to Use?

In fact, some languages embrace multiple paradigms, such as FP + OOP, natively. Examples include Scala (http://scala-lang.org), OCaml (http://ocaml.inria.fr), and F# (http://research.microsoft.com/en-us/um/cambridge/projects/fsharp). Developers can apply features of each paradigm where appropriate, but these languages are inherently more complex.

Alternatively, some projects separate logic into predominantly OO versus FP components. For example, a Web application's UI might be implemented in an OO language such as Ruby, whereas services are implemented in a functional language such as Erlang or Clojure (http://clojure.org). Here, we trade the complexity of a multiparadigm language for the complexity of integrating several, albeit simpler, languages.

An interesting subset of the polyglot trend is the use of multiple languages executing on the same virtual machine. The benefits include easier interoperability between components written in the different languages and minimal overhead for invocations between the languages. For example, languages such as Ruby and Python have been ported to both the JVM (http://ruby.codehaus.org and www.python.org) and the .NET CLR (www.ironruby.net and www.ironpython.com). New languages have been designed exclusively for one or more VMs, such as Groovy (http://groovy.codehaus.org), Scala, Clojure, and F#, even though the JVM was originally designed exclusively for Java. The .NET CLR was designed from the beginning to support multiple languages.

Another class of multilanguage applications are those that combine a "kernel" of functionality, written in a compiled language such as C or C++, with "scripts" written in a higher-level language such as Ruby, Python, Lua (www.lua.org), or Tcl. The compiled kernel provides high performance and access to OS and legacy services, while the scripting language provides higher developer productivity, usually trading off performance. We refer to this architecture as components + scripts = applications. It's a best-of-both-worlds solution, flexible and extensible, yet still providing high performance where necessary.

A well-known historical example is Emacs (www.gnu.org/software/emacs), which combines a C kernel and elisp scripting. The flexibility provided by scripting Emacs in elisp has allowed it to adapt to dramatic technological changes and remain in widespread use for more than 30 years. A recent commercial example of this approach is Adobe Photoshop Lightroom (http://adobe.com/Lightroom3), which combines a C++ kernel with the Lua scripting language (www.lua.org/uses.html#218). Approximately half of the application code is written in Lua. Many C++ game engines are also embedding Lua and using it to implement some functionality.

Domain-specific languages (DSLs) are a kind of polyglot programming. A DSL is a custom, ad hoc language, a form of structured prose that mimics the vocabulary, idioms, and patterns used by experts in a particular domain (www.martinfowler.com/bliki/DomainSpecificLanguage.html). Hence, a DSL minimizes the translation required to implement requirements in executable code. Sometimes, DSLs can even enable end-user programming.

External DSLs are stand-alone languages with their own grammar and custom parser, while internal (or embedded) DSLs are idiomatic dialects of a general-purpose host programming language. Internal DSLs are easier to implement because they don't require special-purpose tools, but external DSLs are more flexible because they aren't limited by a host language's idiosyncrasies. When XML's popularity for representing data experienced rapid growth, many XML DSLs were created for data interchange. Web services standards started including XML message formats, and several sophisticated developer frameworks also emerged to facilitate building small, XML-based languages. More recently, DSLs have become very popular in many programming language communities, such as Ruby’s and Scala’s, where language features make DSL creation relatively easy. DSLs are a special case of language-oriented programming (LOP), which is the practice of creating custom languages for particular problems and programming systems in those languages (www.onboard.jetbrains.com/is1/articles/04/10/lop). Here as well, tools are emerging to make LOP easier for the average developer.

Which Database?

Just as OOP has remained dominant in programming languages and libraries, relational databases have remained dominant for persistence. Several trends have driven renewed interest in alternative strategies. Social networking sites such as Twitter and Facebook maintain massive datasets of relationships, which has driven renewed interest in graph-oriented databases like Neo4J (http://neo4j.org) and FlockDB (http://github.com/...
Internet companies such as Google and Amazon have found it increasingly difficult to scale relational databases for their data volumes, leading to the invention of alternatives that sacrifice some of the ACID (for atomicity, consistency, isolation, and durability) properties in favor of scalability. Collectively, these alternative databases are referred to as NoSQL databases.

**Economics**

Finally, the ever-increasing pressure to shorten development cycles and reduce costs is also driving interest in multiparadigm programming. A well-chosen scripting language or DSL can greatly reduce the amount of code required to implement or modify a feature. Less code means less time spent on a feature in all parts of the software development life cycle. It also means fewer bugs. Sometimes, a DSL enables end-user programming, which completely eliminates developer overhead, at least after the DSL itself is implemented. Similarly, FP code tends to be very concise, and the mathematical underpinnings of FP promote correctness as well as provide better approaches to concurrency.

**Articles in This Special Issue**

This brings us to the articles in this issue, all of which explore different facets of multiparadigm programming.

The well-known Ruby on Rails Web framework uses several internal DSLs to build applications with minimal code and high developer productivity. In “Multi-DSL Applications with Ruby,” Sebastian Günther explores a DSL-based approach to Ruby Web application development that emphasizes feature-oriented programming, in which application development is organized around the development and composition of features. Like Ruby on Rails, he also uses several internal DSLs for the different tiers but emphasizes each feature’s coherence across subsystem boundaries. Developers of multitier applications often struggle to maintain this coherence, especially in larger systems.

Contrast this approach with the work described in “Separation of Concerns and Linguistic Integration in WebDSL,” by Danny M. Groenewegen, Zef Hemel, and Eelco Visser. They also strive for consistency across component boundaries, but in their approach, a universal DSL called WebDSL functions as a coordinator of sublanguages, each of which addresses an individual concern in a typical Web application. The authors show how this approach is particularly useful for cross-cutting concerns, such as maintaining a consistent security strategy across a system’s component boundaries, the kind of problem for which aspect-oriented programming was invented (http://aosd.net).

In “Magic Potion: Incorporating New Development Paradigms through DSLs,” Dragan Djuric and Vladan Devedzic explore a technique of incorporating “parts” of other development paradigms into a host environment that doesn’t support them natively. Using a metaprogramming DSL written in Clojure, they demonstrate the creation of ontological modeling features accessible natively to any other JVM-based language. This approach is attractive for teams that might want to adopt new techniques but are unwilling or unable to adopt new languages and paradigms wholesale.

Departing from the DSL theme, “Streamlining Development for Networked Embedded
Systems Using Multiple Paradigms,” by Christophe Huygens, Danny Hughes, Bert Legaisse, and Wouter Joosen, discusses a common problem in embedded systems: how to tailor the software development process to account for physical, deployment, and other constraints at all stages in the development process, using the example of networked embedded systems. They combine component-based development, aspect-oriented composition, and the effective use of declarative programming to tailor the software life cycle to ensure that all constraints are satisfied on a continuous basis, thereby avoiding costly rework to fix problems that are otherwise discovered later on.

Many application problems are best modeled using constraints, yet constraint-based programming is underutilized. In “Constraint-Based Object-Oriented Programming,” Petra Hofstedt discusses her experience integrating a custom language for constraint-based programming into the Java object model. She compares her results with API alternatives.

Finally, “Multiparadigm Data Storage for Enterprise Applications,” Debasis Ghosh explores the combination of newer NoSQL databases with traditional relational databases in large enterprise applications. Like polyglot programming with languages, he uses polyglot persistence to exploit the complementary features of very different persistence stores.

We conclude the special issue section with an email conversation on multiparadigm programming and the world of software development that we had with Neal Ford and Brian Goetz. Neal originally coined the term polyglot programming, and he has promoted the concept at conferences and in client engagements as an architect at Thoughtworks. Brian is one of the world’s leading experts on the JVM and on lock-based concurrency programming. He’s closely involved in the JVM’s evolution, first at Sun Microsystems and now at Oracle.

As Neal said in our email exchange, “MPP is really just a realization that we’ve been doing it accidentally for a while now.” New approaches in software development usually emerge out of informal and ad hoc practice, as people wrestle with new, but pervasive challenges. When patterns emerge, they’re cataloged, refined, and evangelized. The resurgent interest in MPP reflects the nature of today’s applications and the competitive pressures faced by developers. The challenge is to harness and exploit multiple paradigms. What are the strengths and weaknesses of multiparadigm approaches relative to other technologies? How does an organization evaluate these trade-offs relative to its needs? How should an organization structure its multiskilled teams to develop, deliver, and maintain multiparadigm products? This special issue examines these questions. We encourage the software development community to understand and embrace MPP.

Reference