

Magic Potion: A Metalanguage for Incorporating New Development Paradigms

Dragan Djuric and Vladan Devedzic, *University of Belgrade, Serbia*

Metaprogramming supports a lightweight process to incorporate different programming paradigms in a single development environment that's suitable for small development teams.

Software environments are typically based on a single programming paradigm, such as ontologies, functions, objects, or concurrency. This can limit what developers can represent and how elegant their solutions can be, so today's applications usually involve mixing and matching languages, platforms, and paradigms. However, cross-mappings multiple paradigms and platforms generate an impedance mismatch that increases a solution's complexity.¹ Worse still, even

if your preferred environment requires only a few features from another paradigm, you must typically adopt the whole alien platform to take advantage of them. The alternative of using other languages and tools to implement the features in a way that avoids adding the whole platform is generally at least as difficult.

But a more affordable solution is often possible. We used metaprogramming to incorporate an ontology-modeling paradigm as an embedded domain-specific language (DSL) in our Java-based programming environment. Our approach relies on the use of Clojure,² an emerging language for the Java Virtual Machine (JVM). We also use Modeling Spaces,³ a conceptual framework for studying heterogeneous modeling problems more uniformly.

We wanted a practical way for small development projects with limited resources to include a new paradigm in a host environment. The re-

sult is Magic Potion, a domain-specific metalanguage for incorporating ontological, functional, object-oriented, and concurrent programming paradigms in a single development environment. In this article, we describe its application in our Java ecosystem as a concise, lightweight means for modeling business domains. We also present results from a preliminary evaluation of its implementation advantages and disadvantages. You can access Magic Potion at www.uncomplicate.org/magicpotion.

Multiparadigm Programming versus Metaprogramming

Multiparadigm programming is often useful, but it can be hard to comprehend and implement. It's easier to comprehend when the relationships between paradigm models and platforms are clear. Modeling Spaces and model-driven engineering in general can provide a uniform ground for

Homogeneous Metaprogramming with Clojure

representing paradigms and the relationships relevant to a given project.⁴

It's also easier to use when it's implemented on a single platform with a single language. Metaprogramming can help reduce the number of heterogeneous platforms.⁵ For example, Java doesn't support the ontology paradigm, so to use ontological models, we had to integrate XML-based RDF/OWL⁶ as an external technology through APIs and repositories. If we could support the ontology paradigm on the Java platform in a native way, it could access and be accessed by other parts of a program in the same language without impedance mismatch. The same principle applies to integrating other mainstream paradigms, such as object-oriented, relational, functional, procedural, and concurrent development, as well as specialized paradigms such as those based on rules, fuzzy logic, agents, and neural networks.

However, metaprogramming can also be hard to comprehend and implement. It's easier to use if we can apply it homogeneously—that is, if the language we use for regular programming tasks has metaprogramming facilities. Clojure enables homogeneous metaprogramming (see the sidebar), whereas other JVM languages, such as TXL, augment Java grammar to enable a heterogeneous, ontology-supporting syntax. This is more difficult not only to implement but also to support with Java tools. Again, Modeling Spaces helps identify what to implement and in which context on a specific platform as well as how one paradigm relates to other metamodels.

Incorporating Paradigms

We wanted to be able to select the paradigm best suited to the task at hand, incorporate it into our native environment, and adapt it further to suit our needs. We use the term “incorporate” because we didn't want to implement the whole paradigm. Rather, we wanted to introduce it unobtrusively—just enough to solve a problem elegantly—and blend it with the host language to support multiparadigm programming in a homogeneous environment.

Metaprogramming lets us introduce the paradigm as a homogeneous DSL,⁷ incorporating just the few additional constructs we need. Developing such a DSL must address the following key points:

- **Homogeneity.** Implement the DSL in the native platform and language to lessen the complexity incurred from using multiple paradigms.

Metaprogramming is a process of making *metaprograms* that manipulate other programs, called *object programs*, as data.¹ Typical metaprogramming includes

- dynamic source-code generation, such as SQL expressions and scripts;
- language-extension mechanisms, such as Java annotations;
- metaprogramming languages that extend the host language or create a new one; and
- creation of compilers and similar functions.

Metaprogramming languages can be *heterogeneous*, when the metalanguage is different from the object language, and *homogeneous*, when the metalanguage and the object language are the same. Heterogeneous languages, such as TXL, Stratego/XT, and ML,¹ usually offer more flexibility than homogeneous languages. However, they require specialized knowledge of both metaprogram and object-program internals, which makes them more difficult to learn and use.

Lately, general-purpose languages with metaprogramming capabilities are attracting programmers' attention. They offer more metaprogramming support than mainstream languages such as Java or C#. Ruby, Python, Perl, Lua, and other dynamic languages offer various metaprogramming techniques.

Clojure (<http://clojure.org>) is a recently developed language that compiles directly to Java or Common Language Runtime bytecode. It natively integrates with Java and extends it to support pure functional style, pragmatic facilities for concurrent programming, metaprogramming, and domain-specific languages. Clojure is a homoiconic language—that is, program code is represented as the language's fundamental data type. It supports homogeneous metaprogramming on a Java platform via a macro facility.

Clojure appeared in 2008. Pragmatic Bookshelf published a book on it in 2009,² and three books from major programming literature publishers are scheduled to appear in 2010.

References

1. T. Sheard, “Accomplishments and Research Challenges in Meta-Programming,” *Proc. 2nd Int'l Workshop Semantics, Applications, and Implementation of Program Generation*, LNCS 2196, Springer, 2001, pp. 2–44.
2. S. Hallway, *Programming Clojure*, Pragmatic Programmers, 2009.

- **Minimum additional features.** Implement only the features you need in the host environment to minimize disturbances in the application architecture and development process.
- **Host-platform support.** A homogeneous DSL introduces only a few, nonintrusive constructs, so implement it to be compatible with the host platform's standard tools. Some platforms offer languages with native metaprogramming support.
- **Consistency.** Implement the DSL to make programming with the new paradigm transparent in the host platform.

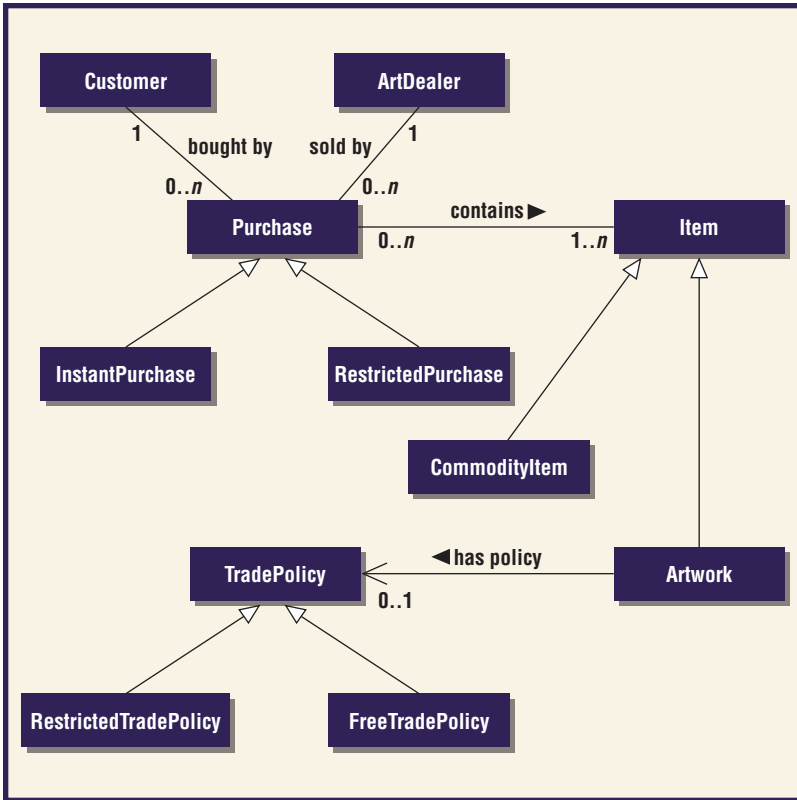


Figure 1. Unified Modeling Language diagram of an art-dealership business domain. Customers purchase items from art dealers either instantly as commodities or under contract restrictions.

- *Clarity.* Create a clear picture of what the new paradigm is good for and why it's needed. We use Modeling Spaces to clarify these issues.
- *Familiarity.* To ensure easier adoption, integrate the DSL in a way that makes the host environment's users comfortable when using it.
- *Tools and services.* Require as simple tools and support as possible because they aren't always readily available in emerging technologies. Also, note whether the tools are open source and free or not.
- *Trends.* Implement the new paradigm and DSL to support current trends and developers' interests and be in line with available literature and resources.

For more information on metaprogramming DSL requirements, see Diomedis Spinelli.⁸

Discovering a Suitable Paradigm

We now explore the integration of a new paradigm into a host environment through a business-domain example.

Art Dealership Domain

Figure 1 shows a UML sketch of a simplified art-dealership business domain. Customers purchase items from art dealers. These items might be commodities that are instantly available or

artwork that's available under trade policies requiring approval from an authority. Such policies are normally part of a much larger domain that has many more business rules and policies to consider and favors the declarative programming style, which lets you freely combine arbitrary predicates.

Domain Modeling Issues

Although the object-oriented approach is widely used to model various business application domains, it suffers from some serious limitations:⁴

- it models behavior, not data semantics;
- it often requires complex processes to model domain tasks that aren't built-in (constraints, complex associations between objects, and so on);
- it's not based on a mathematically sound theory, which leads to ad hoc implementations that are difficult to analyze;
- its concept of inheritance, as implemented in mainstream object-oriented languages, doesn't accord with set theory; and
- its mutable objects aren't suitable for parallel computations.

On the other hand, we can think of business-domain modeling as a kind of knowledge modeling,⁴ an AI field that became popular through the Semantic Web movement.⁶ The Semantic Web is based on ontologies—a descriptive, flexible, and theoretically sound basis for modeling knowledge according to description logics and other mathematical formalisms. However, XML-based RDF/OWL ontologies have some practical drawbacks. They aren't executable, which means they're stored in repositories written in, say, Java and must be accessed and manipulated through repository APIs such as Jena, which pose an infrastructure burden and make them programmer-unfriendly.

We wanted to take what's best from the ontology paradigm—its expressiveness and theoretical basis in description logics—and implement it natively in Clojure to support domain-driven programming that would make the ontology features accessible to all JVM-based programs. With metaprogramming, we can use multiparadigm languages and also shape them to best suit particular tasks, skills, and preferences.

The Magic Potion Metaprogramming Language

Magic Potion introduces the ontology paradigm

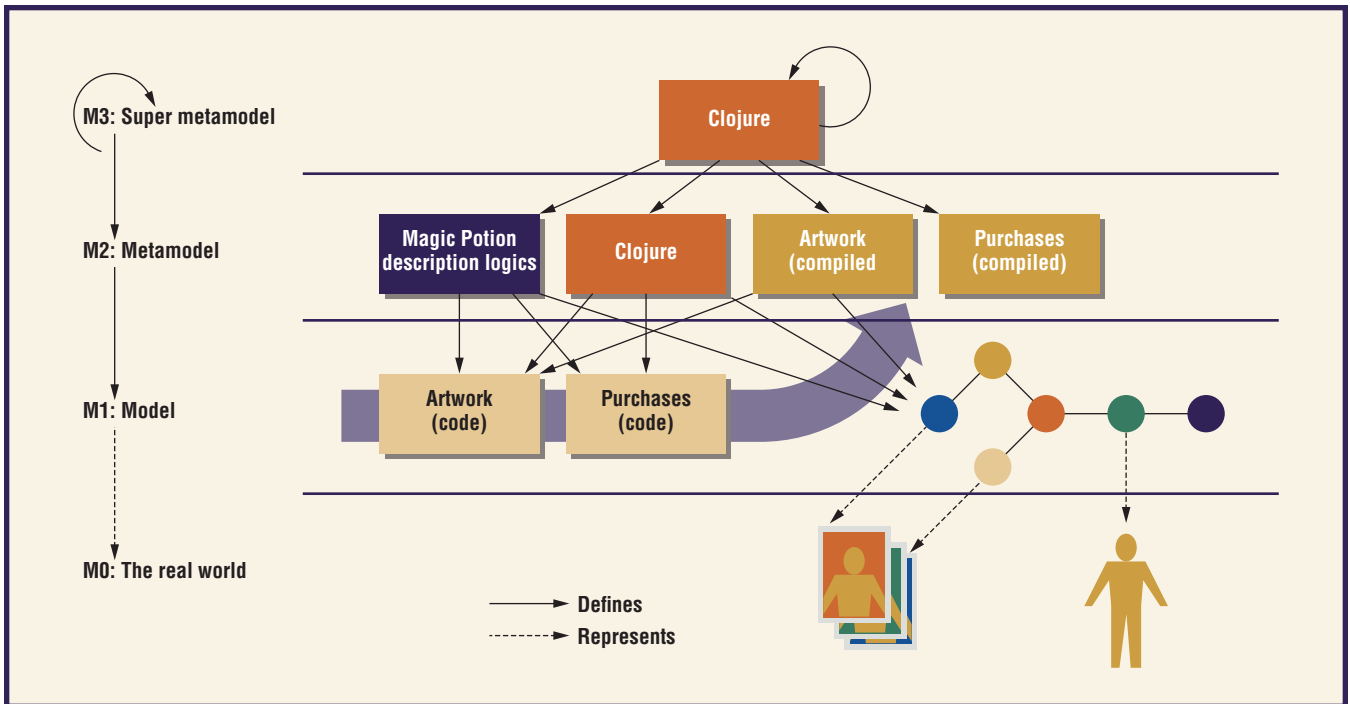


Figure 2. Magic Potion within the Model Driven Architecture. Clojure compiles directly to Java and provides the supermetamodel for building all metamodels, including Magic Potion as a domain-specific metaprogramming language for capturing business-domain models.

for business-domain modeling into Clojure. As Figure 2 shows, we're using Clojure in the Java technical space^{3,4} to build Magic Potion as a DSL for creating other DSLs related to real-world domains. Magic Potion lets us capture the semantics of business processes through ontologies. It seamlessly fits the concepts of rich domain modeling into the concurrent-programming paradigm based on Clojure's Software Transaction Memory. It's practical and easy for software developers to comprehend, requiring little, if any, theoretical knowledge. At the same time, it's formally sound.

In this case, Clojure is a better language choice than Scala or JRuby to meet the requirements for native parallel-programming support and seamless blend into Java's preferred data structures. Other languages might better fit the homogeneity, host-platform support, consistency, and familiarity challenges of other projects.

In the Model Driven Architecture's layer M3 modeling space, Clojure is a supermetamodel, defining all metamodels (programs), even itself. At layer M2, Magic Potion is a Clojure program, a metamodel defined with Clojure functions, macros, and data structures. At layer M1, we use functions and macros defined in Magic Potion to create models that describe abstract domain concepts and their relations. These models (the code

we write) become useful only when we compile them with the Clojure compiler. At that moment, they become minilanguages that can, when run as programs in memory, describe customers who have purchased specific paintings under specific policies on specific dates and for specific prices. Artwork, Purchases, and Customers DSLs become metamodels at layer M2. When executed, they create instances of the respective objects (at M1) that represent real-world artworks, customers, and purchase processes (M0).

The implementation isn't trivial, but it's accessible to any competent Clojure programmer, as the code in Figure 3 shows. The key ingredients are descriptions of *metaconcepts* (concept, role, property, restrictions, inheritance, and so on), *closures* that create functions for each concept that can create and validate instances of that concept, and *macros* that enable smooth integration into Clojure.

Using the Incorporated Paradigm

Ontology properties are independent first-class objects, unlike the attributes or associations of the object-oriented paradigm. Magic Potion lets programmers use description-logics abstractions (concepts, properties, roles, and restrictions) as if they were parts of Clojure. For example, the following code declares two properties, `aname`

```

(defn create-concept [concept-def]
  (let [concept-name (:name concept-def)
        concept-struct (create-struct-deep concept-def)
        validators (create-validators (deep :roles concept-def))
        hierarchy (infer-hierarchy (make-hierarchy) concept-def)
        instance-metadata {:type concept-name
                           ::def concept-def
                           ::hierarchy hierarchy
                           ::struct concept-struct
                           ::validation/validators validators}]
    (ref (fn [& role-entries]
           (with-meta
            (apply struct-map concept-struct
                        (validate validators role-entries))
             instance-metadata))
         :meta {:type ::concept
                ::def concept-def
                ::hierarchy hierarchy
                ::name concept-name
                ::validation/validators validators})))

(defmacro concept [name & params]
  ;; supporting code left out for brevity
  '(let [name-keyword# (to-keyword ~name)
        concept-def# (create-concept-def
                      name-keyword#
                      (sanitize-roles ~roles)
                      (map concept-def ~super))]
      (do
        (def ~(suf-symbol name "?") (is-instance name-keyword#))
        (def ~name (create-concept concept-def#))))))

```

Figure 3.
Implementation code. Magic Potion incorporates ontology properties into the Clojure/Java programming ecosystem.

and `human-name`, and a concept `customer` that uses `human-name` as one of its roles:

```

(property aname [string?])

(property human-name
 :restrictions [(length-between 2 32)]
 :super [aname])

(concept customer
 [human-name])

```

This domain declaration defines the `customer` function we can call to create statements about a customer, such as a valid human name in a string between 2 and 32 characters. (The keywords `:restrictions` and `:super` can enhance readability but aren't usually mandatory.) If we try to create an invalid statement, we get a validation exception with a report listing the unsatisfied restrictions for each property. The application can use this list to produce a nice-looking

error report to the user who supplied the invalid data through, say, a Web form:

```

(customer ::human-name "A")

java.lang.IllegalArgumentException
  ([:user/human-name (length_between)])

```

An attempt to create a customer with valid statements would return its representation as a Clojure persistent map:

```

(customer ::human-name "Jason?Bourne")

{:human-name "Jason?Bourne"}

```

In addition to unqualified restrictions, which apply to a property regardless of the concept in its range, Magic Potion supports qualified restrictions, which apply to a property only in the context of a certain concept. The following declaration of a concept `item` further restricts `aname` only when used as a role of `item`:

```

(concept item
 [(val> aname [(min-length 8) (max-length 256)])])

```

Now we can declare properties that define statements about something that contains `item`, is bought by `customer`, and is sold by `dealer`; and we can use them as roles in `purchase`. As the following code shows, Magic Potion supports arbitrary predicates on all role statements—for example, `min-count` constrains multiplicity:

```

(property contains [item?])

(property bought-by [customer?])

(property sold-by [dealer?])

(concept purchase
 [(ref> bought-by)
  (ref> sold-by)
  (ref*> contains [] [(min-count 1)])])

```

Because these properties' domains aren't simple data types but are instead individuals represented as immutable maps of statements, we defined the role to use Clojure refs instead of direct values. To suit parallel computation better, Clojure uses immutable data structures (in this case, maps). Once we create a map, it represents a snapshot value of an individual that can't be changed. If we used direct values, `purchase` would eternally refer to the

specific customer information, even when that information becomes outdated (for example, when a customer moves to a new address and makes new purchases).

The `ref` function lets us create an immutable statement referencing another individual that can change its immutable value only in the scope of a transaction that's managed through software transactional memory. The `ref>`, `val>`, `ref*>`, and `val*>` functions all create roles—one or many—that use appropriate referencing and multiplicity.

The next few declarations follow the same principles:

```
(concept trade-policy)
```

```
(concept free-trade-policy
 :super [trade-policy])
```

```
(concept restricted-trade-policy
 :super [trade-policy])
```

```
(property has-trade-policy [trade-policy?])
```

```
(concept artwork
 [has-trade-policy]
 [item])
```

```
(concept commodity
 :super [item])
```

```
(concept restricted-purchase
 [(ref> approved-by [authority?])]
 [purchase])
```

Clojure multimethods provide polymorphism based on arbitrary functions, in addition to parameter types. As the newly implemented ontology paradigm natively blends into Clojure, we can take advantage of multimethods for more complex predicates. Depending on the item type (in a general case, on any arbitrary function of an item) our tiny artwork DSL enables creating and validating the statements about different kinds of purchases:

```
(defmulti eligible-for-free-trade? type)
```

```
(defmethod eligible-for-free-trade?
 :commodity [an-item]
 true)
```

```
(defmethod eligible-for-free-trade?
 :artwork [an-item]
 (-> an-item has-trade-policy free-trade-policy?))
```

```
(concept instant-purchase
 [(ref*> contains [eligible-for-free-trade?])]
 [purchase])
```

Clojure offers several ways of calling from Java code, so we can also use the newly created business domain from Java:

```
Var customer = RT.var("purchases", "customer");
Keyword humaname = (Keyword)RT.var("purchases",
 "human-name").invoke();
Object jason = customer.invoke(humanname,
 "Jason?Bourne");
```

We've demonstrated how to implement a new paradigm in the host language. Because Clojure has native support for metaprogramming, so Magic Potion actually consists of functions and macros that don't differ at all from regular Clojure code. Because the ontology paradigm sits on top of Clojure's native data structures, the domain functions create validated business objects as statements in ordinary Clojure maps with additional metadata, and Clojure uses the statements in its usual way.

Evaluating the Metaprogramming Approach

We created Magic Potion to suit our exact needs, so it's considerably simpler and faster than heavyweight ontology solutions, such as Jena. It has exactly the advantages we defined as a goal, and we can live with its disadvantages, such as comparative immaturity, obscurity, and the need for custom maintenance. However, these metrics are mostly subjective, specific to us, and not generally relevant.

Fortunately, we had an opportunity to evaluate this approach with a group of 30 students that attend our MSc course, Software Engineering Tools and Methodologies. Most of the students were active software developers familiar with mainstream languages (Java, .NET, PHP, and so on). More than half of them were familiar with Semantic Web technologies such as OWL and Jena. A few were familiar with alternative languages, such as Ruby and Python, but none were familiar with Clojure and advanced metaprogramming.

During the course, the students first had to learn Clojure and then learn and use Magic Potion to create models for the domains they had worked with previously. Table 1 summarizes our findings from this class regarding the key points we identified earlier.

Magic Potion actually consists of functions and macros that don't differ at all from regular Clojure code.

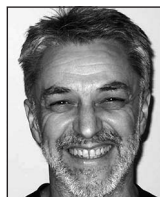
Table 1**Magic Potion evaluation summary form**

Issue	Advantages	Disadvantages
Homogeneous approach	Easy to use; helps understand Clojure	Applicable only in host languages with rich metaprogramming support
Minimum additional features	Most Magic Potion features are frequently used; easy to learn; helps in learning Semantic Web technologies	Doesn't support all features of the original paradigm
Host-platform support	Blends well with the host platform; supported by all regular tools	Some developers might not want to use the original platform
Consistency	Fully consistent with the host language	None
Clarity	Easier to understand, learn, and use	None
Familiarity	Intuitive to the host platform's developers	Advantages limited to the host language
Tools and services	Can be supported by all host-platform tools and services; good for prototyping and experimentation	Applicable languages are still emerging; not yet suitable for mainstream projects
Trends	Advanced developers are interested in such technologies; there is a clear need; recent industry interest in suitable host languages	Not yet in the mainstream; still mostly applicable for prototypes and experiments

About the Authors

Dragan Djuric is an assistant professor at the University of Belgrade, Serbia. His research interests include software engineering and intelligent systems. Djuric received his PhD in information systems from the University of Belgrade. Contact him at dragan@dragandjuric.com.

Vladan Devedzic is a professor of computer science at the University of Belgrade, Serbia. His main research interests include software engineering, intelligent systems, and applications of artificial intelligence techniques to education and healthcare. Devedzic received his **HIGHEST DEGREE?** in **SUBJECT???** from **INSTITUTION???**. Contact him at devedzic@fon.rs.



This approach is still exotic to most mainstream developers. It can be a great fit for exploratory and experimental programming, prototyping, and specialized solutions. The need to use multiple paradigms anyway might bring it closer to more conservative projects, especially once the emerging languages with strong metaprogram-

ming support gain more ground in mainstream projects.

Some developer communities have been implementing new programming paradigms in their languages for decades, but they were usually involved in large-scale efforts that carry additional complexity and require tools and knowledge not common in smaller development environments. Recently revived interest in alternative programming paradigms has led to the development of new languages for mainstream environments, which opens new possibilities for smaller development teams with limited resources.

Homogeneous metaprogramming enables multiparadigm programming on a single platform and language. Embedding a homogeneous DSL in a host language to support specific paradigm features is an alternative to adding a new platform to the environment. It's a useful, general-purpose technique that avoids the complexity of multiparadigm development. It requires less tooling, produces more easily maintainable code, and lets developers stay inside the comfort zone of their preferred environment. ☺

References

1. R. Lämmel and E. Meijer, "Mappings Make Data Processing Go 'Round: An Inter-Paradigmatic Mapping Tutorial," *Generative and Transformation Techniques in Software Eng.*, LNCS 4143, Springer, 2006, pp. 169–218.
2. R. Hickey, "The Clojure Programming Language," *Proc. 2008 Symp. Dynamic Languages*, ACM Press, 2008, article 1.
3. D. Djuric, D. Gasevic, and V. Devedzic, "The Tao of Modeling Spaces," *J. Object Technology*, vol. 5, no. 8, 2006, pp. 125–147.
4. D. Gasevic, D. Djuric, and V. Devedzic, *Model Driven Engineering and Ontology Development*, 2nd ed., Springer, 2009.
5. T. Sheard, "Accomplishments and Research Challenges in Meta-Programming," *Proc. 2nd Int'l Workshop Semantics, Applications, and Implementation of Program Generation*, LNCS 2196, Springer, 2001, pp. 2–44.
6. N. Shadbolt, W. Hall, and T. Berners-Lee, "The Semantic Web Revisited," *IEEE Intelligent Systems*, vol. 21, no. 3, 2006, pp. 96–101.
7. L. Tratt, "Domain Specific Language Implementation via Compile-Time Metaprogramming," *ACM Trans. Programming Languages and Systems (TOPLAS)*, vol. 30, no. 6, 2008, p. 31.
8. D. Spinellis, "Rational Metaprogramming," *IEEE Software*, vol. 25, no. 1, 2008, pp. 78–79.



Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.