developerWorks®

Solving the Expression Problem with Clojure 1.2

Extend preexisting types to new methods, and preexisting methods to new types

Stuart Sierra

December 14, 2010

Clojure expert Stuart Sierra introduces you to new features in Clojure 1.2 that solve the Expression Problem, a classic programming dilemma. *Protocols* let you extend preexisting types to new methods, and *datatypes* let you extend preexisting methods to new types — all without changing the existing code. You'll also see how Java™ interfaces and classes can interact with Clojure protocols and datatypes.

Protocols are a new feature introduced in version 1.2 of Clojure, a dynamic programming language for the JVM. Protocols represent an approach to object-oriented programming that's more flexible than Java class hierarchies, without sacrificing the excellent performance of JVM method dispatch. Protocols — and a related feature, *datatypes* — provide a solution to what is known as the Expression Problem. Solving the Expression Problem makes it possible to extend preexisting types to new methods and extend preexisting methods to new types, all without recompiling existing code.

Clojure basics

This article assumes you're familiar with the basics of writing and running Clojure (and Java) programs. For an introduction to Clojure, see the developerWorks article "The Clojure programming language," and explore the Related topics.

This article describes the Expression Problem, shows some examples of it, and then demonstrates how Clojure's protocols and datatypes can solve it, which simplifies certain programming challenges. You'll also see how you can integrate Clojure's protocol and datatype features with existing Java classes and interfaces.

Many types, one interface

One of Clojure's core features is its generic data-manipulation API. A small set of functions can be used on all of Clojure's built-in types. For example, the `conj` function (short for *conjoin*) adds an element to any collection, as shown in the following REPL session:

```
user> (conj [1 2 3] 4)
[1 2 3 4]
user> (conj (list 1 2 3) 4)
(4 1 2 3)
user> (conj {:a 1, :b 2} [:c 3])
{:c 3, :a 1, :b 2}
user> (conj #{1 2 3} 4)
#{1 2 3 4}
```

Each data structure behaves slightly differently in response to the `conj` function (lists grow at the front, vectors grow at the end, and so on), but they all support the same API. This is a textbook example of *polymorphism* — many types accessed through one uniform interface.

Polymorphism is a powerful feature and one of the foundations of modern programming languages. The Java language supports a particular kind of polymorphism called *subtype polymorphism*, which means that an instance of a type (class) can be accessed *as if* it were an instance of another type.

In practical terms, this means that you can work with objects through a generic interface such as `java.util.List` without knowing or caring if an object is an `ArrayList`, `LinkedList`, `Stack`, `Vector`, or something else. The `java.util.List` interface defines a contract that any class claiming to implement `java.util.List` must fulfill.

The Expression Problem

Philip Wadler of Bell Labs coined the term *Expression Problem* in an unpublished paper that he circulated by email in 1998 (see Related topics). As he put it, "The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts)."

To illustrate the Expression Problem, Wadler's paper uses the idea of a table that has types as rows and functions as columns. Object-oriented languages make it easy to add new rows — that is, new types extending a known interface — as shown in Figure 1:

Figure 1. Object-oriented languages: Easy to add new rows (types)

In Figure 1, each column represents a method in the `java.util.List` interface. For simplicity, it includes only four methods: `List.add`, `List.get`, `List.clear`, and `List.size`. Each of the first four rows represents a class that implements `java.util.List` : `ArrayList`, `LinkedList`, `Stack`, and `Vector`. The cells where these rows and columns intersect represent existing implementations (provided by the standard Java Class Library) of these methods for each of the classes. A fifth row added at the bottom represents a new class you might write that implements `java.util.List`. For each cell in that row, you would write your own implementation of the corresponding method in `java.util.List`, specific to your new class.

In contrast to object-oriented languages, functional languages typically make it easy to add new columns — that is, new functions that operate on existing types. For Clojure, this might look like Figure 2:

Figure 2. Functional languages: Easy to add new columns (functions)

Figure 2 is another table, similar to Figure 1. Here, the columns represent functions in Clojure's standard collection API: `conj`, `nth`, `empty`, and `count`. Similarly, the rows represent Clojure's built-in collection types: `list`, `vector`, `map`, and `set`. The cells where these rows and columns intersect represent existing implementations of these functions provided by Clojure. You can add a new column to the table by defining a new function. Assuming your new function is written in terms of Clojure's built-in functions, it will automatically be able to support all the same types.

Wadler's paper was talking about statically typed languages such as the Java language. Clojure is dynamically typed — the specific types of objects do not need to be declared, or known, at compile time. But that doesn't mean Clojure gets a free pass on the Expression Problem. In fact, it changes almost nothing. Being dynamically typed doesn't mean that Clojure doesn't *have* types. It just doesn't require you to declare all of them in advance. The same problem of extending old functions to new types, and new functions to old types — extending Wadler's table in both directions — remains.

A concrete example

The Expression Problem isn't just about abstract types such as lists and sets. If you've spent a significant amount of time working with object-oriented languages, you've probably run into examples of the Expression Problem. This section provides a concrete, albeit simplified, real-world case.

Suppose you work in the IT department at WidgetCo, a mail-order office-supply company. WidgetCo has written its own billing and inventory-management software in the Java language.

WidgetCo's products are all described by a simple interface:

```
package com.widgetco;

public interface Widget {
    public String getName();
    public double getPrice();
}
```

For each unique type of widget manufactured by WidgetCo, the IT department's programmers write a class that implements the `Widget` interface.

An order from one of WidgetCo's customers is implemented as a list of `Widget` objects, with one extra method to compute the order's total cost:

```
package com.widgetco;

public class Order extends ArrayList<Widget> {
    public double getTotalCost() { /*...*/ }
}
```

Everything works fine at WidgetCo until the company is bought out by Amagalmated Thingamabobs Incorporated. Amalgamated has its own custom billing system, also written in the Java language. Its inventory is centered around an abstract class, `Product`, from which specific product classes are derived:

```
package com.amalgamated;

public abstract class Product {
    public String getProductID() { /*...*/ }
}
```

At Amalgamated, products do not have fixed prices. Instead, the company negotiates with each customer to deliver a certain quantity of items at a given price. This agreement is represented by the `Contract` class:

```
package com.amalgamated;

public class Contract {
    public Product getProduct() { /*...*/ }
    public int getQuantity()    { /*...*/ }
    public double totalPrice()  { /*...*/ }
    public String getCustomer() { /*...*/ }
}
```

After the merger, your new bosses at Amagalmated assign you the task of writing a new application to generate invoices and shipping manifests for the merged company. But there's a catch: the new system must work with the existing Java code used for inventory management at both WidgetCo and Amalgamated Thingamabobs — the `com.widgetco.Widget`, `com.widgetco.Order`, `com.amalgamated.Product`, and `com.amalgamated.Contract` classes. Too many other applications depend on that code to risk changing it.

You have just met the Expression Problem.

Potential "solutions"

Object-oriented languages offer a number of possible approaches to solving the Expression Problem, but each one has its drawbacks. You've probably encountered examples of each of these techniques.

Inheritance from a common superclass

The traditional object-oriented solution to problems of this sort is to leverage subtype polymorphism — that is, inheritance. If two classes need to support the same interface, they should both extend the same superclass. In my example, both `com.widgetco.Order` and `com.amalgamated.Contract` need to produce invoices and manifests. Ideally, both classes would implement some interface with the necessary methods:

```
public interface Fulfillment {
    public Invoice invoice();
    public Manifest manifest();
}
```

To do this, you'd need to modify the source code for `Order` and `Contract` to implement the new interface. But that was the catch in this example: you can't modify the source code to those classes. You can't even recompile them.

Multiple inheritance

Another approach to the Expression Problem is *multiple inheritance*, in which one subclass can extend many superclasses. You want a generic representation of a purchase that can be either a `com.widgetco.Order` or a `com.amalgamated.Contract`. The following pseudocode shows what this might look like if the Java language had multiple inheritance:

```
public class GenericOrder
        extends com.widgetco.Order, com.amalgamated.Contract
        implements Fulfillment {

    public Invoice invoice() { /*...*/ }

    public Manifest manifest() { /*...*/ }
}
```

But the Java language doesn't support multiple inheritance of concrete classes, for a good reason: it leads to complex and sometimes unpredictable class hierarchies. The Java language *does* support multiple inheritance of interfaces, so if `Order` and `Contract` were both interfaces, you could make this technique work. But regrettably, the original authors of `Order` and `Contract` weren't farsighted enough to base their design on interfaces. Even if they were, this wouldn't be a real solution to the Expression Problem, because you can't add the `Fulfillment` interface to the existing `Order` and `Contract` classes. Instead, you have created the new `GenericOrder` class, which has the same problems as wrappers, described next.

Wrappers

Another popular solution is to write *wrappers* around classes whose behavior you want to modify. The wrapper class is constructed with a reference to the original class, and it forwards methods to that class. The Java wrapper for the `Order` class might look like this:

```
public class OrderFulfillment implements Fulfillment {
    private com.widgetco.Order order;

    /* constructor takes an instance of Order */
    public OrderFulfillment(com.widgetco.Order order) {
        this.order = order;
    }

    /* methods of Order are forwarded to the wrapped instance */

    public double getTotalCost() {
        return order.getTotalCost();
    }

    /* the Fulfillment interface is implemented in the wrapper */
    public Invoice invoice() { /*...*/ }

    public Manifest manifest() { /*...*/ }
}
```

`OrderFulfillment` is a wrapper around `Order`. The wrapper class implements the `Fulfillment` interface I described earlier. It also copies and forwards methods defined by `Order`. Because `Order` extends `ArrayList<Widget>`, a correct implementation of the wrapper class would also need to copy and forward all the methods of `java.util.ArrayList`.

Once you create another wrapper for `com.amalgamated.Contract` that also implements `Fulfillment`, you've satisfied the requirements of your task — but at the cost of increasing complexity. Wrapper classes are tedious to write. (`java.util.ArrayList` has more than 30 methods.) Worse, they break certain behaviors you expect of your classes. An `OrderFulfillment`, although it implements the same methods as `Order`, is not really an `Order`. You can't access it through a pointer declared as an `Order`, nor can you pass it to another method expecting an `Order` as an argument. By adding a wrapper class, you have broken subtype polymorphism.

Even worse, wrapper classes break *identity*. An `Order` wrapped in an `OrderFulfillment` is no longer the same object. You can't compare an `OrderFulfillment` to an `Order` with the Java `==` operator and expect it to return `true`. If you try to override the `Object.equals` method in `OrderFulfillment` such that it is "equal" to its `Order`, you've broken the contract of `Object.equals`, which specifies that equality must be symmetric. Because `Order` knows nothing about `OrderFulfillment`, its `equals` method will always return `false` when passed an `OrderFulfillment`. If you define `OrderFulfillment.equals` such that it can return `true` when passed an `Order`, you've broken the symmetry of `equals`, causing a cascade of failures in other classes, such as the built-in Java collection classes, that depend on that behavior. The moral of the story is: don't mess with identity.

Open classes

The Ruby and JavaScript languages have helped to popularize the idea of *open classes* in object-oriented programming. An open class isn't limited to the set of methods that were implemented when it was defined. Anyone can "reopen" the class at any time to add new methods, or even replace existing methods.

Open classes permit a high degree of flexibility and reuse. Generic classes can be extended with functionality specific to different places where they are used. Groups of methods implementing a particular aspect of behavior can be gathered together in a *mix-in* that is added to any class that needs that behavior. If this article's example were written in Ruby or JavaScript, you could simply reopen the `Order` and `Contract` classes to add the methods you need.

The downside of open classes, aside from their nonexistence in most programming languages (including the Java language), is the lack of certainty brought about by their very flexibility. If you define an `invoice` method on a class, you have no way of knowing that some other user of that class won't define a different, incompatible method also named `invoice`. This is the problem of *name clashes*, and it's difficult to avoid in languages with both open classes and no namespacing mechanism for methods. There's a reason this technique is known as "monkey patching." It's simple and easy to understand, but it's almost guaranteed to cause problems later.

Conditionals and overloading

One of the most common solutions to the Expression Problem is plain old if-then-else logic. Using conditional statements and type checks, you enumerate every possible case and deal with each appropriately. For the invoice example, in the Java language, you could implement this in a static method:

```
public class FulfillmentGenerator {
    public static Invoice invoice(Object source) {
        if (source instanceof com.widgetco.Order) {
            /* ... */
        } else if (source instanceof com.amalgamated.Contract) {
            /* ... */
        } else {
            throw IllegalArgumentException("Invalid source.");
        }
    }
}
```

Like the wrapper class, this technique fulfills your requirements, but it comes with its own disadvantages. The chain of if-else blocks gets messier, and slower, the more types you must handle. And this implementation is *closed*: After you compile the `FulfillmentGenerator` class, you can't extend the `invoice` method to new types without editing the source code and recompiling. The conditional solution to the Expression Problem is really no solution at all, just a hack that will lead to more maintenance work in the future.

In this case, you could avoid the conditional logic by overloading the `invoice` method for different types:

```
public class FulfillmentGenerator {
    public static Invoice invoice(com.widgetco.Order order) {
        /* ... */
    }

    public static Invoice invoice(com.amalgamated.Contract contract) {
        /* ... */
    }
}
```

This accomplishes the same thing and is more efficient than the conditional version, but it's just as closed: You can't add new implementations of `invoice` for different types without modifying the source of `FulfillmentGenerator`. It also becomes unpredictable in the face of inheritance hierarchies. Suppose you wanted to add an implementation of `invoice` for a *subclass* of `Order` or `Contract`. Only a Java language expert could tell you which version of the method would actually be invoked, and it might not be the one you want.

Real solutions to the Expression Problem in the Java language *do* exist (see Related topics), but they are much more complex than you would want to tackle to solve a simple business problem. In general, all of the nonsolutions I've presented here fail because they conflate types with other things: inheritance, identity, or namespaces. Clojure treats each issue separately.

Protocols

Clojure 1.2 introduced protocols. While not a new idea — computer scientists were researching similar ideas in the 1970s — Clojure's implementation is flexible enough to solve the Expression Problem while preserving the performance of its host platform, the JVM.

Conceptually, a protocol is like a Java interface. It defines a set of method names and their argument signatures, but no implementations. The invoice example might look like Listing 1:

Listing 1. The `Fulfillment` protocol

```
(ns com.amalgamated)

(defprotocol Fulfillment
  (invoice [this] "Returns an invoice")
  (manifest [this] "Returns a shipping manifest"))
```

Each protocol method takes at least one argument, commonly called `this`. Like Java methods, protocol methods are invoked "on" an object, and that object's type determines which implementation of the method is used. Unlike the Java language, Clojure requires `this` to be declared as an explicit argument to the function.

Protocols differ from interfaces in that their methods exist, as ordinary functions, the moment they are defined. The Listing 1 example defines Clojure functions named `invoice` and `manifest`, both living in the `com.amalgamated` namespace. Of course, those functions don't have any implementations yet, so calling them will just throw an exception.

Protocols allow you to provide implementations for their methods on a case-by-case basis. You *extend* the protocol to new types, using a function named `extend`. The `extend` function takes a *datatype*, a protocol, and a map of method implementations. I'll explain datatypes in the next section, but for now, just assume that datatypes are ordinary Java classes. You can extend the `Fulfillment` protocol to work on the old `Order` class, as shown in Listing 2:

Listing 2. Extending the `Fulfillment` protocol to work on the `Order` class

```
(extend com.widgetco.Order
  Fulfillment
  {:invoice (fn [this]
              ... return an invoice based on an Order ... )
   :manifest (fn [this]
               ... return a manifest based on an Order ... )})
```

Notice that the map you pass to the `extend` function maps from keywords to anonymous functions. The keywords are names of the methods in the protocol; the functions are implementations of those methods. By calling `extend`, you are telling the `Fulfillment` protocol, "Here is the code to implement these methods on the `com.widgetco.Order` type."

You can do the same thing for the `Contract` class, as shown in Listing 3:

Listing 3. Extending the `Fulfillment` protocol to work on the `Contract` class

```
(extend com.amalgamated.Contract
  Fulfillment
  {:invoice (fn [this]
              ... return an invoice based on a Contract ... )
   :manifest (fn [this]
               ... return a manifest based on a Contract ... )})
```

And with that, you have polymorphic `invoice` and `manifest` functions that can be called on either an `Order` or a `Contract` and that will do the right thing. You've satisfied the difficult part of the Expression Problem: you've added the new `invoice` and `manifest` functions to the preexisting `Order` and `Contract` types, without modifying or recompiling any existing code.

Unlike in the open-classes approach, you have not changed the `Order` and `Contract` classes. Nor have you prevented other code from defining its own `invoice` and `manifest` methods with different implementations. Your `invoice` and `manifest` methods are namespaced within `com.amalgamated`; methods defined in other namespaces will never clash.

Clojure, being a Lisp, uses *macros* to simplify some complex or repetitive syntax. Using the built-in `extend-protocol` macro, you could write all your method implementations in one block, as shown in Listing 4:

Listing 4. Using the `extend-protocol` macro

```
(extend-protocol Fulfillment
  com.widgetco.Order
    (invoice [this]
      ... return an invoice based on an Order ... )
    (manifest [this]
      ... return a manifest based on an Order ... )
  com.amalgamated.Contract
    (invoice [this]
      ... return an invoice based on a Contract ... )
    (manifest [this]
      ... return a manifest based on a Contract ... ))
```

This macro code expands out to the same two calls to `extend` shown in Listing 2 and Listing 3.

Datatypes

Protocols are a powerful tool: they effectively give you the ability to insert new methods into existing classes, without name clashes and without modifying the original code. But they only solve half of the Expression Problem, the "new columns" of Wadler's table. How do you add "new rows" to the table in Clojure?

The answer is *datatypes*. In the object-oriented landscape, datatypes fill the same role as classes: they encapsulate *state* (fields) and *behavior* (methods). However, mainstream object-oriented languages such as the Java language tend to conflate the different roles that classes can fill in object-oriented design. Clojure's datatypes are split into two distinct roles.

Structured data: `defrecord`

One use for classes is as *containers for structured data*. This is where almost every object-oriented programming textbook starts: you have a `Person` class with fields like `Name` and `Age`. Whether the fields are accessed directly or through getter/setter methods is immaterial; the class itself fills the role of a `struct` in C. Its purpose is to hold a set of logically related values. JavaBeans are examples of classes used to hold structured data.

The problem with using classes for structured data is that each class has its own distinct interface for accessing that data, typically through getter/setter methods. Clojure favors uniform interfaces, so it encourages storing structured data in maps. Clojure's map implementations — `HashMap`, `ArrayMap`, and `StructMap` — all support the generic data-manipulation interface I discussed at the beginning of this article. But maps lack some of the features expected of structured data: they have no real "type" (without added metadata) and, unlike classes, they can't be extended with new behavior.

The `defrecord` macro, introduced along with protocols in Clojure 1.2, can be used to create containers for structured data that combine the features of maps and classes.

Returning to this article's running example, suppose the management of Amalgamated Thingamabobs Incorporated decides it's time to unify the purchasing processes of its customers. From this point forward, new product orders will be represented by a `PurchaseOrder` object, and the old `Order` and `Contract` classes will be phased out.

A `PurchaseOrder` will have "date," "customer," and "products" properties. It must also provide implementations of `invoice` and `manifest`. As a Clojure `defrecord`, it looks like Listing 5:

Listing 5. The `PurchaseOrder` datatype

```
(ns com.amalgamated)

(defrecord PurchaseOrder [date customer products]
  Fulfillment
    (invoice [this]
      ... return an invoice based on a PurchaseOrder ... )
    (manifest [this]
      ... return a manifest based on a PurchaseOrder ... ))
```

The vector following the name `PurchaseOrder` defines the datatype's *fields*. After the fields, `defrecord` allows you to write in-line definitions for protocol methods. Listing 5 implements only one protocol, `Fulfillment`, but it could be followed by any number of other protocols and their implementations.

Under the hood, `defrecord` creates a Java class with `date`, `customer`, and `products` fields. To create a new instance of `PurchaseOrder`, you call its constructor in the usual way, supplying values for the fields, as shown in Listing 6:

Listing 6. Creating a new `PurchaseOrder` instance

```
(def po (PurchaseOrder. (System/currentTimeMillis)
                        "Stuart Sierra"
                        ["product1" "product2"]))
```

Once the instance is constructed, you can call the `invoice` and `manifest` functions on your `PurchaseOrder` object, which uses the implementations provided in the `defrecord`. But you can also treat the `PurchaseOrder` object like a Clojure map: `defrecord` automatically adds the methods necessary to implement Clojure's map interface. You can retrieve fields by name (as keywords), update those fields, and even add new fields not in the original definition — all using Clojure's standard map functions, as in the following REPL session:

```
com.amalgamated> (:date po)
1288281709721
com.amalgamated> (assoc po :rush "Extra Speedy")
#:com.amalgamated.PurchaseOrder{:date 1288281709721,
                                :customer "Stuart Sierra",
                                :products ["product1" "product2"],
                                :rush "Extra Speedy"}
com.amalgamated> (type po)
com.amalgamated.PurchaseOrder
```

Notice that when a record type is printed, it looks like a map with an extra type tag in front. That type can be retrieved with Clojure's `type` function. The fact that maps and record types obey the same interface is especially convenient during development: you can start with ordinary maps and switch to record types when you need the extra features, without breaking your code.

Pure behavior: `deftype`

Not all classes represent structured data in the application domain. The other kind of class typically represents objects specific to an implementation, such as collections (for example, `ArrayList`, `HashMap`, `SortedSet`) or values (for example, `String`, `Date`, `BigDecimal`). These classes implement their own generic interfaces, so it doesn't make sense for them to behave like Clojure maps. To fill that role, Clojure offers a variant of datatype for things that aren't records. The `deftype` macro has the same syntax as `defrecord`, but it creates a bare-bones Java class, without any map-like features. Any behavior of the class must be implemented as protocol methods. You would typically use `deftype` to implement new kinds of collections, or to define completely new abstractions around protocols that you specify.

One feature that is deliberately left out of datatypes is the ability to implement methods that are not defined in any protocol. A datatype must be completely specified by its fields and the protocols it implements; that guarantees that functions invoked on a datatype will always be polymorphic. Any datatype can be substituted by another datatype as long as they implement the same protocols.

Interacting with Java code

Over the past several months, you have managed to sneak some Clojure into the production systems of Amalgamated Thingamabobs Incorporated. Because it was compiled into JAR files, the other programming teams never even noticed. But then another team of Java-only developers needs to build another invoice generator that is compatible with your Clojure code. You start to talk to them about Clojure, Lisp, and the Expression Problem, and their eyes glaze over.

Finally, you say, "No problem, just make your Java classes implement this interface," and you show them this:

```
package com.amalgamated;

public interface Fulfillment {
    public Object invoice();
    public Object manifest();
}
```

Clojure has already compiled the protocol you defined earlier into JVM bytecode representing this interface. In fact, *every* Clojure protocol is also a Java interface with the same name and methods. If you're careful with naming (that is, don't use any characters in your method names that aren't valid in the Java language) then other Java developers can implement the interface without ever knowing that it was generated from Clojure.

The same is true of datatypes, although their generated classes look a bit less like what a Java developer would expect. The `PurchaseOrder` datatype defined in Listing 5 generates bytecode for a class like this:

```
public class PurchaseOrder
    implements Fulfillment,
               java.io.Serializable,
               java.util.Map,
               java.lang.Iterable,
               clojure.lang.IPersistentMap {

    public final Object date;
    public final Object customer;
    public final Object products;

    public PurchaseOrder(Object date, Object customer, Object products) {
        this.date = date;
        this.customer = customer;
        this.products = products;
    }
}
```

The datatype's fields are declared `public final` in keeping with Clojure's policy of immutable data. They can only be initialized through the constructor. You can see that this class implements the `Fulfillment` interface. The other interfaces are the foundations of Clojure's data-manipulation APIs — they would be absent in a datatype created with `deftype` instead of `defrecord`. Notice that datatypes can implement methods of any Java interface, not only those generated by Clojure protocols.

Clojure has other forms of Java interop that produce code that's more like idiomatic Java — but less like idiomatic Clojure — when that's the only option.

Conclusion

The Expression Problem is a real, practical problem in object-oriented programming, and most common solutions to it are inadequate. Clojure protocols and datatypes provide a simple, elegant solution. Protocols allow you to define polymorphic functions over preexisting types. Datatypes let you create new types that support preexisting functions. Together they let you extend your code in both directions, as shown in Figure 3:

Figure 3. Clojure: Easy to add new columns (protocols) and rows (datatypes)

Figure 3 shows the final version of the table, adapted from Figure 1 and Figure 2. The columns represent any functions or methods that already exist. Rows represent classes (Java) and datatypes (Clojure) that already exist. The intersection of those rows and columns represents all preexisting implementations, which you don't want or need to change. You can add a new Clojure datatype as a new row at the bottom of the table, and a new Clojure protocol as a new column at the right side of the table. Furthermore, your new protocol can be extended to your new datatype, filling in the new bottom-right cell.

Solutions to the Expression Problem beyond those I have described here are available. Clojure itself has always had *multimethods*, which provide an even more flexible variety of polymorphism (albeit one that performs less well). Multimethods are similar to *generic functions* in other languages such as Common Lisp. Clojure's protocols also bear a resemblance to Haskell's *type classes*.

Every programmer will encounter the Expression Problem sooner or later, so every programming language has some sort of answer to it. As an interesting experiment, pick any programming language — one you know well or one you are just learning — and think about how you would use it to solve a problem like the example presented in this article. Can you do it without modifying any existing code? Can you do it without breaking identity? Can you be sure you're safe from name clashes? And, most important, can you do it without creating an even bigger problem in the future?

Related topics

- Clojure: Visit the Clojure website and check out the Protocols and Datatypes pages.
- Stuart Halloway on Clojure: Listen to this podcast with a Clojure committer to learn more about Clojure and why it's rising rapidly in popularity.
- The Expression Problem: Read Philip Wadler's paper.
- Clojure's Solutions to the Expression Problem: View video and slides from a presentation by Chris Houser at Strange Loop 2010.
- Clojure Protocols: Take a look at slides presented by Stuart Halloway at the International Software Development Conference 2010.
- "The Clojure programming language" (Michael Galpin, developerWorks, September 2009): Get started with Clojure, learn some of its syntax, and take advantage of the Clojure plug-in for Eclipse.
- *Practical Clojure* (Luke VanderHart and Stuart Sierra, Apress, 2010): Check out this book's chapter on datatypes and protocols.
- Type polymorphism and Polymorphism in object-oriented programming: Read Wikipedia's articles about polymorphism, including subtype polymorphism.
- The Expression Problem Revisited: Take a look at a paper, slides, and example code that show real solutions to the Expression Problem in the Java language, by Mads Torgersen, University of Aarhus, Denmark.
- Clojure: Download Clojure 1.2.