

Inq: The Argument for a Reified Application Framework

Tom Sanders

Inqwell Ltd

tom.sanders@inqwell.com

Abstract

The overwhelming popularity of object oriented techniques and languages promotes their use in the implementation of business systems almost without question. This has led to more than one generation of OO frameworks for common services such as transaction management and long-term persistence.

However, over the complete life-cycle of design, implementation, test/rework and maintenance, experience can show the benefits are doubtful and some question the consensus [Hickey].

The theory presented here has been developed into a procedural language and execution environment known as Inq [www.inqwell.com]. In turn, Inq has been used to great effect to develop the Synthetics trading system Xylinq [www.xylinq.com]

1. Introduction

The accepted approach for writing software systems has always been to directly translate application problem issues into the chosen programming language. To set a historical context, where third generation languages are concerned such as C or Cobol, this entailed a separate data and procedural decomposition. Such languages support a small set of features like user data type definitions and arrays.

With the advent of object oriented languages two key features are added:

- Data Encapsulation - user defined data types and the functions performed on them are now held together and the data hidden from public view. With ad-hoc access to data members prevented, the overall system is more tolerant to alterations of the user defined types;
- Polymorphism - the programming language now supports the notion of families of related user defined types where the specific type to which a function is applied need not be known until run-time. Family members can be manipulated through a common base interface.

Fourth generation languages are a misnomer and are, in fact, third generation languages with built in database access add-ons. For the remainder of this discussion we consider only OO systems as they have popularly superseded procedural ones.

OO practises supposedly heralded many benefits in system maintenance. It should be possible to re-use code more readily by "extending the family" using the language feature of inheritance. The principle difference with OO advances is that, if the families are well designed, the system can evolve more easily and with little or no effect on the existing code base. However, system design and implementation is a complex analysis of entities, relationships, functionality and groupings. The problem of how to define these families in the first place remains an intractable one. How sound the design is depends on at least:

1. how well the problem is (or can be) specified;
2. the volatility of the problem domain;
3. subjective views of the OO designers.

Point 2 above is very difficult to quantify since, put simply, it is not possible to predict the future with 100% accuracy. If we concede that our system will be ill-specified and evolve then we must accept that it is not possible to arrive at a set of abstractions representing an initial application and expect that:

1. these abstractions can be adapted and extended as the problem domain does likewise *and*
2. those changes will not affect existing interfaces.

Such changes are likely to cut across the typical layers of GUI, application logic and data.

Consider also the effect of changing an interface. All participants must be made aware of the change by a recompilation of their modules. This may invoke a complete system rebuild and redistribute if changes occur across multiple layers. It is a problem which has always plagued systems and is often made worse, not better, by OO languages, since inheritance is itself defining an interface. We are then drawn down the path of design patterns to mitigate these problems, which are well documented elsewhere [Gamma et al].

Suffice to say that, even with some experience and the best intentions, any product will be a compromise that gets progressively worse over time. This paper argues that we shift that compromise away from the problem domain and into the implementation technology and proposes that systems are implemented in a more loosely typed, meta-data driven environment. Applications are expressed more in terms of the framework's capabilities and less in those of the chosen implementation language.

2. Example - Tabular Data Display

A common low-level problem is to render some user-defined data structure on to a table. It doesn't matter whether this table is a GUI display or just formatted text as the issues are the same. How might we address this?

A typical OO approach might be to offer an interface of the form:

```
interface TableModel
{
    int    getColumnCount();
    int    getRowCount();
    Object getValueAt(int rowIndex, int columnIndex);
}
```

We then state that any user defined type to be so rendered implements this interface to present the underlying data:

```
public class PositionData
{
    private float _value;
    private String _contractRef;

    .
    .
    .

    public float getValue()
    {
        return _value;
    }

    public String getContractRef()
    {
        return _contractRef;
    }
}

public class PositionTableModel implements TableModel
{
    private PositionData[] _rows;

    public int getColumnCount()
    {
        return 5;    // or whatever
    }

    public int getRowCount()
    {
        return _rows.length;
    }

    public Object getValueAt(int rowIndex, int columnIndex)
    {
        if (columnIndex == 0)
            return new Float(_rows[rowIndex].getValue());
        else
            .
            .
    }
}
```

Here, the 'framework', as such, is the class `PositionTableModel` and the interface `TableModel`. There are a number of issues worth pointing out:

1. Every application domain class, `PositionData` in the above example, wishing to render itself to a table must have a corresponding implementation of `TableModel`. This is a lot of repetitive code of the form `getValueAt(...)`.
2. This is only one way we might want to represent data. What about a graphing function? It, too, will most likely provide its own framework of interfaces and methods.
3. At least any changes to `PositionData` will not cause problems but what if we want the same class to be rendered to different tables, one showing more detail, for example? Then we must implement distinct (or at least a family of) `PositionTableModel` classes. We see the requirement arising for a significant amount of code to do essentially the same task. Furthermore this example stops before considering the issue of actually formatting the cell values.

3. Same Data, Many Meanings

Point 3 above, is a specific case of the more general application issue of what does data actually mean? The answer is often not clear cut. Consider a class representing a single investment banking trade. We might want to look at this data in a number of ways:

1. For profit/loss calculations;
2. for submitting to a regulatory exchange;
3. for entry into a back office system.

In recognising this we might place all necessary methods in one class. This supports the best encapsulation of the data but makes the class's code footprint large. Instead we might make a basic `Trade` class and wrap it inside other classes which each support the necessary functionality we wish to group. However, this introduces coupling between the `Trade` class and all its separate domain wrappers.

It is said, *Information is Power*. While the initial set of methods in our `Trade` class might be acceptable, it is likely to get larger as more departments within our organisation demand particular views on our data, let us say for accounts purposes, counter party statistics or corporate actions processing. As our data is put to greater use, so we extend our classes or build the necessary bridges. We will very likely impact on our existing installed base or reproduce essentially similar classes in the new application domains.

4. Common Object Services

There are many common services we would like to bestow on our application objects. Examples are often:

1. save and extract objects from backing store;
2. create, mutate and destroy objects with concurrency and transaction safety;
3. distribute objects between front-end clients and back end servers;
4. raise events for member value change or particular method invocation.

Many tools are aimed at addressing these common services however each has its own learning curve and must be integrated into the local development environment. They often create as many new problems as those in bespoke development they sought to solve. For example, XML dialects are introduced which themselves have to be maintained, or specific mapping interfaces must be implemented.

The common thread underlying all these problems is that, in order to support a given service, some bridge code must be generated or written and maintained, or a tool's configuration and semantics properly understood. Concerning interface changes, it is extremely unlikely that these can be contained, with maintenance implications for distributed systems.

5. Reification

By means of reification something that was previously implicit, unexpressed and possibly inexpressible is explicitly formulated and made available to conceptual (logical or computational) manipulation. [Wikipedia].

The foregoing discussion and examples show the use of OO implementation language features

1. class to define application types;
2. interface to define bridges;
3. member and method access in bridge implementations.

Reifying these concepts, that is transferring them out of compiled-in information and into run-time configuration data, offers several benefits and solutions. Consider a slot based approach. Instead of using the implementation language to specify user defined data types in the application domain, we instead use it to implement a general purpose associative array or *map*. We use configuration data supplied externally to define the contents of a given use of this map. Such a map might look something like:

```
class Map extends Any
{
    public Map(Descriptor d)
    public void setValue (String label, Any value) throws UnknownMember
    public Any getValue (String label) throws UnknownMember
    .
    .
    .
    private HashMap _contents;
}
```

Here, `_contents` holds a dictionary of names associated with values. These values are class types, themselves derived from a base `Any`:

```
class AnyFloat extends Any {...}
class AnyString extends Any {...}
```

and so on. Readers may recognise this class structure as the *Composite* pattern from [Gamma et al]. Typical mutator methods now become `setValue("contractRef", ref);`. `Descriptor` is not shown but is intended to act as data to initialise the number and type of elements in `_contents` and their labels.

Returning to our example `PositionData` class and its corresponding `PositionTableModel`, this now changes:

```

public class Formatter {...}

public class AnyTableModel implements TableModel
{
    // A vector of maps each of which represents
    // a PositionData equivalent
    private Vector _matrix;

    private Vector _columnMap; // specify the desired labels and
                                // their order

    private Vector _formatters; // specify widths, precisions etc

    public AnyTableModel(Vector a) { _matrix = a; }

    public int    getColumnCount()
    {
        return _columnMap.size();
    }

    public int    getRowCount()
    {
        return _matrix.size();
    }

    public Object getValueAt(int rowIndex, int columnIndex)
    {
        Map row = _matrix[rowIndex];

        return row.getValue(_columnMap[columnIndex]);
    }
}

```

What are the differences in these two approaches?

1. Because there are no longer specific members as such, all the code needed to render a Map now exists in the single AnyTableModel implementation.
2. Even if we alter the layout and labels of _contents within the particular Map, provided that _columnMap can still be reconciled against it everything still works. No interfaces or compiled-in object layouts have changed so no rebuilds are necessary. This could even happen at run time by signalling all instances of the Map representing PositionData to change their _contents containers.
3. We can now display PositionData (or anything else) in any way we choose by configuring AnyTableModel instances with the appropriate _columnMap and _formatters data.

This technique has reified the language concept of member access. In doing this we have been able to separate concerns more cleanly than the original table display example. What we have sacrificed is the sure knowledge that member access can succeed since we are assuming that _contents contains at least the labels that are listed in _columnMap.

6. Reifying Class Definitions

When a Map is constructed we supply a Descriptor instance to set up the internal slots. Descriptor is therefore reifying the language feature of class definition. If we have a list of Descriptors held against labels then we can request that any 'class' we have configured into the system be constructed by a single factory method supported in our 'class catalog' singleton.

In static OO, factories require participant classes to provide a method like newInstance:

```
class Foo
{
    public static Object newInstance() { return new Foo (); }
    .
    .
    .
}
```

We then set up a hash table for functions like Foo.newInstance and catalog these against labels.

```
class TheFactory
{
    public Object buildObject (String reqdClass) throws UnknownClass;
    public void addToFactory (String newClassLabel, FuncPtr f);
    private HashTable _factoryList;
}
```

We still, therefore, have a list but in the concrete case the list is tied to all the concrete classes like Foo and can't readily be extended at run-time because we are limited to the universe of compiled-in (or run-time available) classes.

7. Object Distribution Issues

The point has been made that interface changes are made many times worse when objects become distributed. Adopting a general *Composite* pattern means that, like the table display function, the method of distribution becomes merely another service which can be modelled within its own class hierarchy. This must be preferable to methods where such common services intrude into areas of code that ought to be exclusive to the application domain.

Traditional uses of systems such as CORBA and RMI promote complex interfaces which are then tied to implementation classes either by inheritance or code generation. Once a given distribution mechanism has been chosen, or even a particular vendor within that mechanism, it can be difficult to change or support bridges to other policies like DCOM should this later become necessary.

One of the benefits of these technologies is that the range of classes and services is extended beyond one's own systems and applications. Given the interface definition and a server location, we can access third-party systems. However, there will be little or no control available to users of these systems and an interface published to a large client base necessarily makes it difficult for the vendor to change that interface other than by continual extension.

Conversely, with Maps, the interface becomes very simple and any use of distribution technology becomes completely separated. It would even be possible to support multiple distribution technologies simultaneously, the choice being determined by the client at connect time. Changes can occur to what is passed across the interface by altering the contents of a Map but the interface itself remains unchanged. Once the data has been transported to the client, that client can manipulate it in any way desired.

8. Relationships

All OO systems not only define class hierarchies, data content and functionality but also document relationships in the design. At the implementation level, relationships can take a number of forms:

1. as method arguments;
2. as data members - aggregation;
3. by inheritance.

In a reified system we can model aggregation using a containment hierarchy.

Using a Map as an arbitrary container (as opposed to a preconfigured one that represents a domain type), we can construct ad-hoc groupings according to the relationships and usage we would like to prevail in a given context.

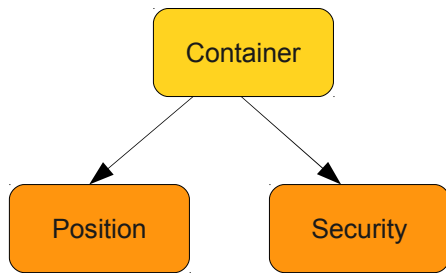


Fig 1 *a 1:1 Aggregation*

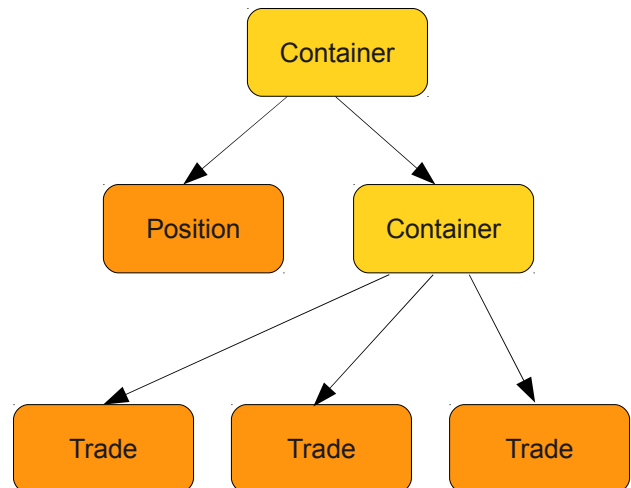


Fig 2 *A 1:n Aggregation*

This is analogous to data members, however in a reified system decisions about what aggregations are useful to the application are made much later than in OO designs, often as the need arises.

What about inheritance? Inheritance in the application domain is probably the most subjective issue during system design. It is also the most tightly coupling of the relationship mechanisms in the OO method - in statically typed OO languages inheritance is programmed into the system and cannot be changed at run-time. In other words, inheritance is something more specific than aggregation, the latter being more appropriate in many cases, and certainly easier to alter during the maintenance phase.

If the problem is in any way subject to change then it quickly becomes impossible to use inheritance in the domain context. It is all too easy to forget that if you can't say the relationship is truly *is a* then you shouldn't be applying inheritance and its use is often small-scale, or only at the level of technical implementation.

9. Reifying Methods

To reify methods, consider the interface definition below:

```
interface Func extends Any
{
    Any doit(Any a);
}
```

The argument `a` represents the context against which `Func` implementations execute, that is the container structure that must satisfy the implementation's references within it.

We can implement various "building block" functions in the implementation language and combine these as required. Consider a function that references a node within a containment hierarchy:

```
public class NodeRef implements Func
{
    String _tokens[];

    public NodeRef(String ref)
    {
        // ref is of the form "a.b.c..." to navigate the containment hierarchy
        _tokens = tokenise(ref);
    }

    Any doit(Any a)
    {
        // Apply tokens of "ref" to the context argument "a"
        // to resolve the successively deeper children

        int i = 0;

        while (a instanceof Map && i < _tokens.length())
        {
            Map m = (Map)a;
            a = m.get(_tokens[i++]);
        }

        return a;
    }
}
```

Both `Func` and non-`Func` types, such as `Map` and numeric values, inherit from `Any`. Generally, we want to execute a network of `Func` instances until a non-`Func` is yielded:

```
static Any execExpr(Any root, Any arg)
{
    Func f = null;
    while (arg instanceof Func)
    {
        f = (Func)arg;
        arg = f.doit(root);
    }

    return arg;
}
```

9.1 A Summation Function

Given a root node (or a reference to it) and a reference to a value within each child of the root, return the sum:

```
public class Sum implements Func
{
    private Any _sumRoot;
    private Any _childRef;

    public Sum (Any sumRoot, Any childRef)
    {
        _sumRoot = sumRoot;
        _childRef = childRef;
    }

    Any doit(Any a)
    {
        // Evaluate the root node to sum beneath
        Map sumRoot = (Map)execExpr(root, _sumRoot);

        // The "+" operator (not shown)
        Add add = new Add();

        Any result = new Value(0);

        Iter i = sumRoot.iterator();
        while(i.hasNext())
        {
            // Loop over the children of sumRoot
            Any child = i.next();

            // Resolve the reference from each child
            child = execExpr(child, _childRef);

            // Add the current child to the running total
            add.setOp1(result);
            add.setOp2(child);
            result = add.evalOperator();
        }
    }
    return result;
}
```

9.2 A Square Root Function

Given a value (or a function that evaluates to a value), return its square root:

```
public class Sqrt implements Func
{
    private Any _value;

    public Sqrt (Any value)
    {
        _value = value;
    }

    Any doit(Any a)
    {
        // Evaluate the reference
        Value square = (Value)execExpr(a, _square);

        // Take the square root
        return new Value(Math.sqrt(square.doubleValue()));
    }
}
```

Given the above examples we can create a Func network to evaluate the square root of the result of a summation:

```
Any sqRootOfSeries = new Sqrt(new Sum(new NodeRef("my.series"),
                                     new NodeRef("a.b.c")));
```

Returning to the absence of polymorphism, classes implementing the Func interface can be applied to as many different types of Map and Map structures as we like. As long as the Map we invoke it on can satisfy the content requirements then the Func network is applicable. This allows re-use of code although it is not polymorphism because the method code is always the same - it does not vary with the class the method is applied to.

Its arguable that this form of type compatibility is still powerful as it doesn't require any relationship between the compatible Maps at all. A true reification of inheritance and polymorphism remains unaddressed by this paper. However, it is the author's belief that with the flexibility provided by Maps and the ability to aggregate them that reification of inheritance is not required.

10. Towards A New Execution Environment

This paper has put forward the argument that moving away from using language features for the direct expression of problem domain entities, leading to function networks operating on arbitrary structures at run-time promotes:

1. the separation of common services and application code;
2. systems that are able to evolve as the problem does.

What sort of things does this approach accommodate?

10.1 Instance Identity and Caching

The use of Map to represent entity instances allows the Descriptor that defines its content to designate one or more of the slot values as primary key fields. The value of these fields determine the instance's identity in the application domain.

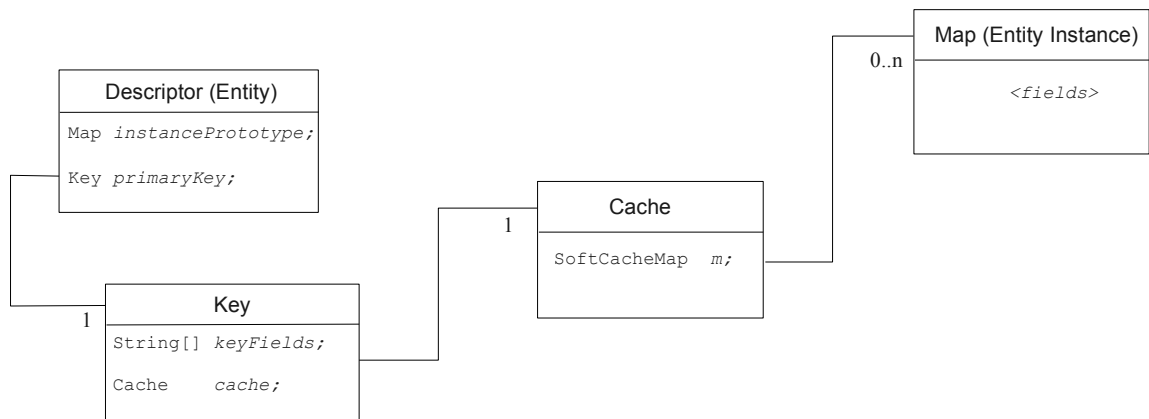


Fig 3 Instance Identity

The primary key fields define a unique, immutable composite value for each instance. Having defined the primary key fields the environment is then able to manage a cache of loaded instances against their primary key.

10.2 Relational Database Bindings

The assertion that the application of OO methods is of little benefit except in a highly constrained problem space is perhaps upheld by the continuing prevalence of relational databases and only niche uses of object ones. The Map and Descriptor classes described above readily lend themselves to relational database persistence.

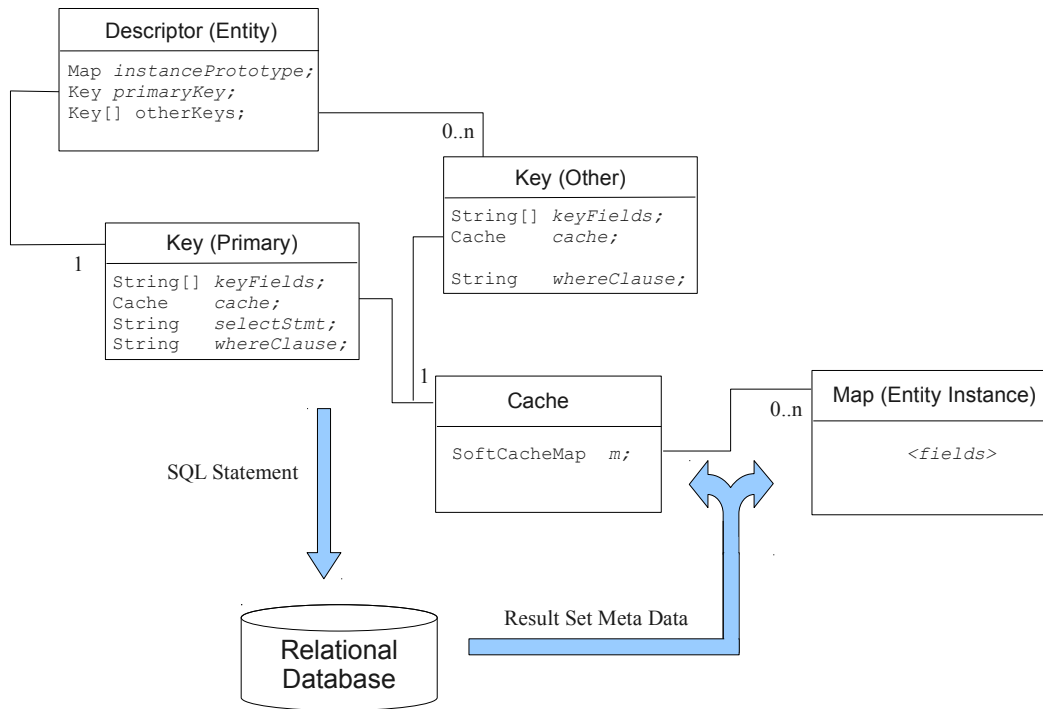


Fig 4 Relational Database Bindings

We can define a *select* statement in the primary key (whose *where* clause applies the primary key value) and any number of other keys with an appropriate *where* clause for look-ups required by the application. Using the meta-data available in the database result set, columns can be matched to the fields of an instance prototype and the primary key used to determine whether the instance is (from another query) already in the cache.

10.3 Transactions and Events

The use of a cache and the reification of identity through the primary key value allows the environment to ensure that a given entity instance is represented by a single Map instance. That is, concurrent processes within a Server environment will all operate on the same physical object, placed into a containment hierarchy of their choosing.

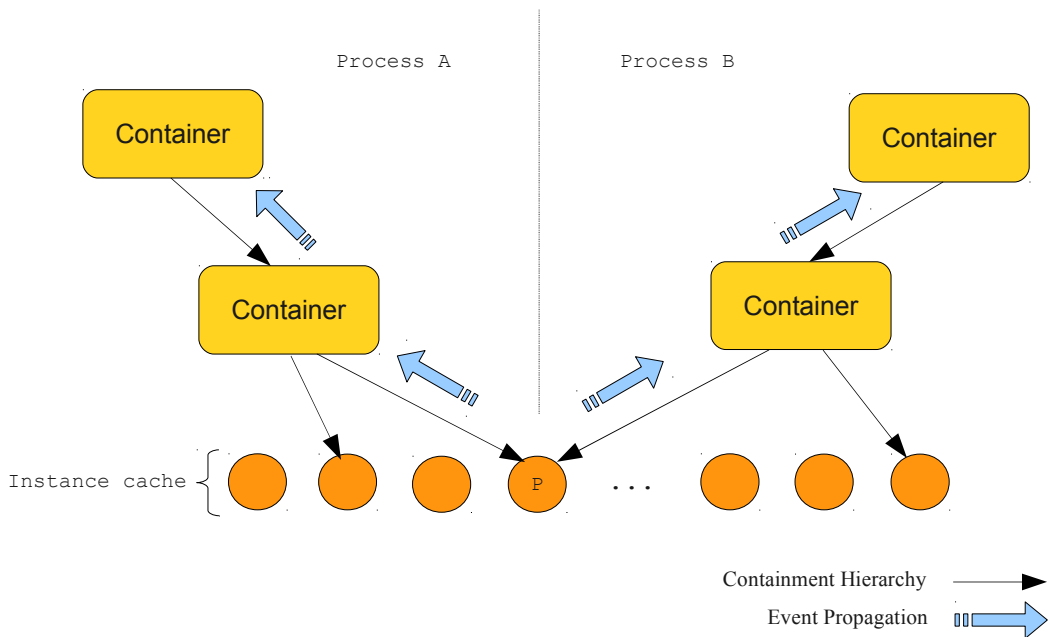


Fig 5 *Event Propagation*

Figure 5 shows two processes A and B that both hold a reference to instance P returned from the cache. If either process modifies a field within P the environment can raise an event that is propagated through all containment hierarchies of P.

Maps are easy to copy and having reified references with the NodeRef class, a clone private to the mutating process's transaction can be taken as fields are modified. Such private instances can then be managed by the process's transaction, with an appropriate locking policy of the shared public instances as required.

10.4 Automatic Support for Model-View-Controller

Consider a map at path c containing a homogeneous set of instances. The map supports vector access so its contents can be rendered as rows in a graphical table. Fields of the instances define the columns of the table model described in section 5.

The graphical table is listening to events emitted from c . If instance P is accessed from c by path k then events raised on mutation of P will have the path $c.k$

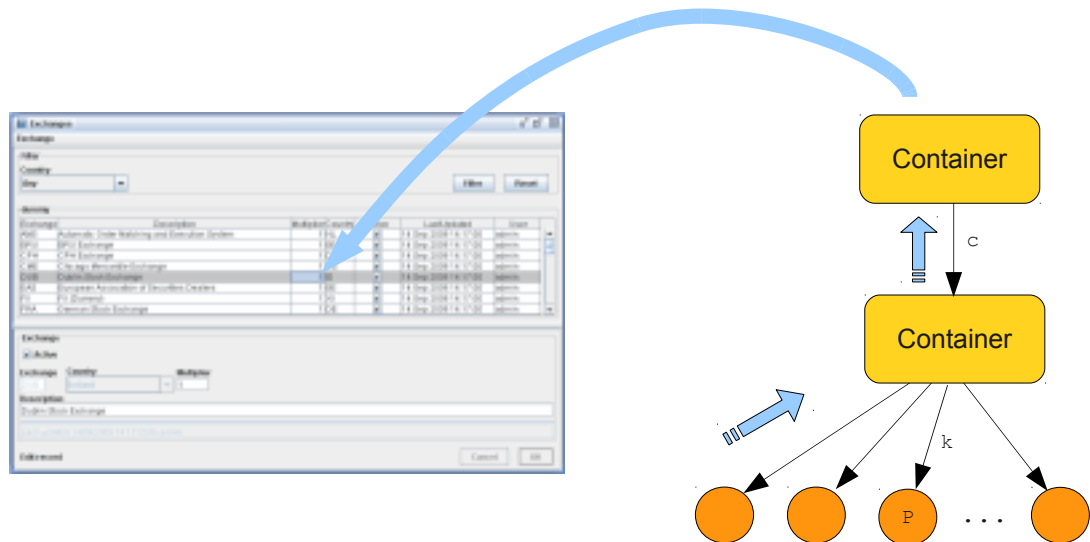


Fig 6 Automatic MVC

We can further configure the graphical table to listen for events matching the path $c.*$ where $*$ represents any child path from c and if the event carries the fields being mutated, the environment can dispatch only events relevant to the model-view relationship between the table and the node space. Similarly, a table that permits editing is able to operate in the controller role and update fields in the data structure.

In an environment that uses maps rather than native language features to represent application types, together with the other reification techniques outlined, all the concepts described in this section can be implemented in a framework ahead of time and completely separately from the problem domain.

11. Inq - A Procedural Language

Having adopted this approach however, it becomes clear, and from [Matzel, Bischofberger] that the implementation language is unsuitable as a way to express applications. Considering the example given earlier, assuming `my.series` can be resolved under the node context we would have:

```
Any sqRootOfSeries = new Sqrt(new Sum(new NodeRef("my.series"),
                                     new NodeRef("a.b.c")));

Any sqrt = sqRootOfSeries.doit(context);
```

Expressing a function network in this way leads to unreadable code and there is little point in having a dynamic environment yet using a compiled one to wire it up. If instead a parser is developed a better syntax can be defined:

```
any sqrt = sqrt(sum($this.my.series, $this.a.b.c));
```

where `$this` is resolved by the environment to be the *current execution node* in a process's node space. Furthermore a syntax is required for defining the problem domain types, to create the environment's Descriptors. The parser can be run at any time to modify the environment's types and function network. The development of this language, *Inq*, is discussed at [www.inqwell.com].

The end result is a procedural language creating (or operating on) a node space that defines a particular execution context. The conception of a new language expressly for the problem domain, running in an environment that provides the required common services, necessarily separates these two things completely. That they *cannot* coexist at the language level prevents any possible pollution of one by the other. Common services in the environment are ultimately reusable and, while their semantics must be taken into consideration, the *Inq* language allows the developer to focus wholly on the problem domain.

12. References and Related Material

Matzel, Bischofberger: *The Any Framework. A Pragmatic Approach to Flexibility* UBILAB, Union Bank of Switzerland, 1996 [http://www.ubilab.org/publications/print_versions/pdf/coots96.pdf]

Eggenschwiler, Gamma: *ET++ SwapsManager: Using Object Technology in the Financial Engineering Domain* UBILAB, Union Bank of Switzerland, 1992 [http://www.ubilab.org/publications/print_versions/pdf/swaps-oops1a92.pdf]

Gamma, Helm, Johnson and Vlissides: *Design Patterns: Elements of Reusable Design* Addison-Wesley, 1995

Birrer, Eggenschwiler: *Frameworks in the Financial Engineering Domain An Experience Report* UBILAB, Union Bank of Switzerland, 1993 [http://www.ubilab.org/publications/print_versions/pdf/bir93b.pdf]

Keller, Coldeway: *Relational Database Access Layers* Software Design and Management GmbH Munich, 1997 [<http://www4.informatik.tu-muenchen.de/proj/arcus/TUM-19746/2.5.ps.gz>]

Riehle: *Patterns for Encapsulating Class Trees*, Union Bank of Switzerland, 1996 [<http://dirkriehle.com/computer-science/research/1995/plop-1995-trading.html>]

Jain, Widdorf, Schmidt: *The Design and Performance of MedJava A Distributed Electronic Medical Imaging System* Department of Computer Science, Washington University, 1997 [<http://www.cs.wustl.edu/~schmidt/PDF/MedJava.pdf>]

Kleinoder, Golm: *MetaJava: An Efficient Run-Time Meta Architecture for Java™* Friedrich-Alexander University, Germany, 1996 [<http://www4.informatik.uni-erlangen.de/Publications/pdf/Kleinoeder-Golm-MetaJava-IWOOOS.pdf>]

Hickey: *Are We There Yet? A Deconstruction of Object Oriented Time* JVM Language Summit, 2009 [<http://wiki.jvmlangsummit.com/images/a/ab/HickeyJVMSummit2009.pdf>]