

# Leveraging Quality

## *Migrating a Mature, High-Level GUI Framework to Java*

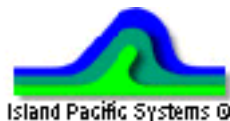
**Brian Schultheiss**  
Island Pacific Systems  
brians@islandpacific.com

**Mark L. Fussell**  
ChiMu Corporation  
mark.fussell@chimu.com

The value of a programming language is measured by how well it helps developers produce solutions. A young language, like Java, usually incurs a significant penalty in value from its initial deficiency of good high-level libraries and frameworks as compared to more mature languages. But Java has been amazingly successful and a major cause of this success is the rapid and effective leveraging of quality work from other OO languages and environments. The effort of developers to migrate good designs and code into Java enabled the language to make a giant leap forward.

Island Pacific Systems used Java to build The Eye™, an OLAP (on-line analytical processor) for its retailing data warehouse product. As part of this project, the Island Pacific team encountered significant deficiencies in the existing Java libraries for producing a clean UI and Domain layering. Ultimately the team decided to migrate the design of a mature, high-level GUI framework from Smalltalk over into Java. The resulting framework increased the value of Java as a tool to Island Pacific with a relatively small investment of time.

Within this report, we will describe the context, process, and results of this migration experience. Overall, the Island Pacific Systems team found Java to be very “receptive” to good designs and patterns from other OO languages even though Java and its libraries are relatively young. Leveraging the high quality work done in other OO languages was extremely useful and effective in solving Island Pacific’s project needs.



### Island Pacific Systems

19800 MacArthur Boulevard, Twelfth Floor  
Irvine, CA 92612

Phone: +1 (949) 476-2212

Fax: +1 (949) 476-0177

[www.islandpacific.com](http://www.islandpacific.com)



1220 N. Fair Oaks Ave, #1314  
Sunnyvale, CA 94089

Phone: 408 734-9068

Email: [info@chimu.com](mailto:info@chimu.com)

[www.chimu.com](http://www.chimu.com)

# Introduction

---

Island Pacific Systems Corporation provides an enterprise software suite called I3™ for the retail industry. The I3™ product was initially written in RPG and runs on the IBM AS/400 platform. The capabilities, reliability, and scalability of the AS/400 made it an excellent platform and allowed I3™ to grow tremendously in functionality. But over time, the advantages of a modern GUI interface, the development benefits of a more sophisticated language than RPG, and the potential for multiple deployment environments caused Island Pacific to investigate moving to a modern, OO language and a cross-platform development environment. In 1997, Island Pacific began evaluating Java™ as a new development tool.

As an initial project, Island Pacific chose to migrate The Eye™ to Java. The Eye™ is an OLAP (on-line analytical processor) that provides decision support capabilities on top of the I3™ Data Warehouse. The Eye™ has several aspects that make it an ideal test-bed for evaluating Java: (1) it was relatively separable from the rest of the application\*, (2) it could significantly benefit from modern GUI capabilities†, (3) it had serious performance and scalability requirements, and (4) it had complex code, which was difficult to maintain in RPG.

At the time of this paper, we are in the third phase of development. The first phase introduced Java into Island Pacific and verified that Java's performance on a client machine was acceptable‡. The second phase focussed on making a commercially releasable version of The Eye™ in Java and on activities that would increase the long-range maintainability of the Java code base and the system design. The third phase continued this focus on maintainability, scalability, and design. Although the goals of the second and third phases were similar, the Island Pacific team took significantly different approaches to reach these goals. Here we will describe the specifics of migrating the design of a high-level GUI framework from another language (Smalltalk) instead of building that functionality on top of low-level Java capabilities or relying on IDE (Integrated Development Environment) behavior.

## Goals

During the second phase of development, the Island Pacific team began focusing on long-range maintainability, scalability, and general design quality. Where the first phase had a "Make it work" focus, the second phase was attempting to "Get it right"§. In terms of GUI capabilities and its relationship to the Domain model, we knew we wanted to:

1. Have a good, clean, layering between the GUI and the Domain
2. Keep business logic out of the GUI
3. Standardize and automate interfacing between GUI and Domain objects
4. Handle type and value conversions required between the GUI and the Domain
5. Capture and report errors in a reasonable fashion
6. Support significant amounts of multi-threading to make the GUI responsive when other operations were delayed
7. Maintain integration with a GUI builder

## Initial Attempts

Although conceptually simple, implementing a good UI-to-Domain layer separation on top of the capabilities available within Java was not easy. Although Java comes with a GUI framework, both the initial AWT and the more sophisticated Swing framework are low-level: they focus on the GUI itself more than the GUI's relationship to other layers in the system. JavaBeans provide more sophisticated object

---

\* The Eye™ needs to use the output of other programs within I3™, but does not need to directly interface to most of those programs.

† The Eye™ includes a spreadsheet metaphor, so there are many ways to utilize GUI capabilities like color, fonts, and so on.

‡ Assuming continued improvements to Java, JVMs, and computing hardware.

§ It was also necessary to "Make it fast", as anyone using Java in 1997 and 1998 will attest to.

capabilities like properties and change events, but these are not fully integrated with the rest of the Java libraries, do not have a full framework perspective, and they have serious scalability issues if not limited. Finally, IDEs (Integrated Development Environments) provide their own conception of how a GUI connects to the rest of the application, but these were not as complete, maintainable, and scalable as we had hoped.

During the first phase, we had no particular best approach. Different developers tried different approaches and the outside mentoring we received at the time did not provide a full architectural standard and training in this area. This was fine as a learning process, but we realized we needed to standardize on an approach that achieved the above goals.

During our second development phase we focused on this standardization and required functionality. A core element of our approach revolved around a heavy utilization of VisualAge<sup>®</sup> for Java<sup>™</sup> as both a general development tool, and as the GUI builder & GUI-to-application connector. At the time we thought that the ease of which VAJ allowed us to create beans and connect property change events to property updates was invaluable. Likewise, we were enamoured by the ability to “draw” connections from the property of a domain object to a property of a visual object. This enabled the IDE to generate a significant amount of code automatically and resulted in a great productivity boost for us. On top of the VisualAge capabilities, we created a UI-to-Domain synchronization framework that enabled us to multi-thread time consuming activities and give a responsive application.

Although leveraging VisualAge’s “drawing” capabilities appeared to be very useful at first, we found that it did not scale well for our actual application needs. It could become quite difficult to draw the required kinds of visual connections and it proved to be cumbersome to build reusable components that correctly separated UI and domain. Even when successfully separated, the code that was generated by the IDE was hard to trace and very expensive in runtime performance. Ultimately the approach proved to be both incomplete and a bit too complex to reach our desired goals.

## Migration Process

---

The next approach we used for reaching our GUI framework goals was to leverage an existing, mature framework that provided most of the functionality we required. It was not required that this framework be written in Java, but it needed to be complete and tested. We did not want to only take a few good ideas, ideally we wanted to migrate a whole concept that fit well together. This would reduce the risks of subtle incompatibilities, poor scalability, or poor maintainability. On top of this base we could add customizations from our own experiences and the projects specific needs.

To reach the GUI layering goals identified above required adding a single main architectural concept: a Domain Presentation layer. The Domain Presentation layer would be responsible for connecting a very general GUI layer to a transactional, GUI independent, Domain layer. Both the GUI layer and the Domain layer can be conceptually understood independently of each other and the presentation layer handles the coupling between them.

### *Starting Frameworks*

A framework that provided this capability in VisualWorks<sup>®</sup> Smalltalk was the **Domain Presentation** framework described by Tim Howard in *The Smalltalk Developer’s Guide to VisualWorks* [Howard 95]. This framework was mature at the time of the book and had been subsequently tested & augmented by an Island Pacific team member’s experiences on several Smalltalk projects after that publication date. Overall, at least seven years of trial and modification had gone into the framework in its Smalltalk form. So the primary goal was to take advantage of this complete framework by migrating it to Java.

Unfortunately, we could not just individually migrate this one framework. It was a high-level framework and had strong dependencies on capabilities not available in Java. Smalltalk has enormous amounts of functionality in its core libraries, and when developing an application in Smalltalk you leverage both obvious and subtle capabilities of these libraries to make your task much easier. Although this is a good

behavior for developing in Smalltalk, it means migrating a Smalltalk framework to a different environment will require migrating similar capabilities from the core libraries as well.

Specifically, the Domain Presentation concept required a substrate of a general **ValueModel** framework. ValueModels\* can be thought of as “active variables” or “notifying holders of objects”. Whenever a change occurs to a ValueModel (e.g. the holder has a new value), the Model’s dependents/listeners will be notified. This general ValueModel behavior was a critical element to providing a common, flexible interface between a general event-oriented UI and a general Domain object model. Although the Java libraries had the beginnings of such concepts as part of Swing and JavaBeans, they were not on the level of generality and integration required. So an implementation of the ValueModel concept would also be required.

An even more basic substrate was a formalized concept of **Functors** within Java (see [ChiMu-1]). A Functor is “An object that models an operation” [Firesmith+E 95]. Specifically in Java, a Functor can be defined as a simple single-operation interface. This allows a client to be given a completely general interface for calculating a value or doing an action and the object implementing the interface can support it in however simple or complex a manner as is required. This concept was specifically necessary to allow very flexible “plugability” of ValueModels with each other, but is also useful for other general purposes.

For this report we will focus on the migration of the Domain Presentation framework itself and any of augmentation of the ValueModel capabilities to work with Domain Presentation and the Java GUI†.

## *Transformations*

Transforming the Domain Presentation framework into Java required four kinds of activities: (1) converting between the Smalltalk language to the Java language, (2) moving to the Java environment and especially the GUI libraries, (3) customization of the frameworks for Island Pacific design needs, and (4) improving terminology‡. Although these activities will be described in an order, they were not performed sequentially. Also, none of these changes are code-to-code transformations§: both subtle and obvious design changes needed to be made throughout and were then implemented in a natural way in Java.

## **Converting to the Java language**

The initial process of migration involves converting from Smalltalk, a dynamically typed language, to Java, a statically typed language\*\*. At its simplest, this conversion simply requires creating Java interfaces for all the major roles that a Smalltalk object plays within the frameworks and then suitably including this type information within the behavior desired. The reality is much more complex. The use of interfaces as the common glue throughout Java is a common and ChiMu development standard (See [ChiMu-2] and [Coad+M 96]), so this itself caused no difficulties. But Smalltalk supports a very fine-granularity of inter-object protocol, a large amount of behavior within Object itself, and the ability to cleanly “augment”†† existing classes. Altogether these would lead to an explosion of interfaces and classes within Java, and the Java code would still have difficulties reproducing the full functionality.

Fortunately the goal was not to migrate general Smalltalk capabilities to Java and retain that generality. The goal was to migrate the capabilities and architecture of several frameworks for a specific goal. For example, all Smalltalk objects have general Notifier and Listener capability. Although possibly useful, migrating this verbatim to Java is impossible because we do not control “java.lang.Object”. Further that

\* See [Woolf-1], [Liebs+R 92], [Krasner+P 88], and many of the other Smalltalk and Design Patterns books in the references.

† The migration of ValueModel and Functors capabilities to Java had already been accomplished by previous ChiMu work and will not be discussed here.

‡ Improvements to terminology included making it more main-stream with Design Pattern names, making it more compatible with other Java terminology (Notifier & Listener), or simply choosing a more intuitive name for someone with no Smalltalk background.

§ The main developer neither knew Smalltalk nor ever saw Smalltalk code as part of the migration.

\*\* This could also be called migrating from an optimistic, fine-grained typed language to a pessimistic, course-grained typed language. See [Fussell-1] for a discussion of these terms.

†† An “augmentation” is behavior added to an existing class from a new source control unit (e.g. a new “package” or “project”). This was first available through the Envy product (now part of VisualAge) but is now common to multiple development environments.

full migration was simply not necessary: we only cared about particular areas (e.g. ValueModels) having this notification capability so we should focus on that.

Probably the most important feature of the Java language to enable some type of migration of some very generic structures was the runtime typecheck (also called a “cast” but it behaves quite differently from a converting cast). This enabled objects to migrate through the generic “tunnels” that connected the UI to the Domain and to have the type information “restored” to the object at the other side. For example, a ValueModel provides a very generic interface:

```
public Object getValue();
public void setValue(Object newValue);
```

This generality enables all object properties to be treated similarly by the GUI independently of how complicated they are underneath, but at some point the ‘newValue’ and the result of the ‘getValue’ need to be verified and “unmasked” into the true types expected. For example, to respond to a change in the number of decimal places for a currency, we would need to “unmask” ‘newValue’ into an Integer and ultimately into an ‘int’:

```
public void setValue( Object newValue ) {
    if( shouldUpdate( getCurrency().getDecimalPlaces(),newValue ) ) {
        getCurrency().setDecimalPlaces( ((Integer) newValue ).intValue() );
        fireValueChanged();
    }
}
```

The full code handles any problems through a “ProblemRecorder” before reaching this section, so the above cast is guaranteed to succeed.

Partially these typechecks are caused by the relatively weak type system of Java: it does not have covariant return types or parameterized types. But even with a fuller type system, many of the capabilities of the frameworks would have been difficult to reproduce. The loophole of a dynamic typecheck in Java made it possible to support the generality that the Domain Presentation framework and supporting frameworks desired.

## Moving to the Java environment

Moving to the Java environment had many areas tractable but a

### *Integrating with the Java GUI frameworks*

The largest issue with moving the Domain Presentation concept into the Java environment was that Java’s GUI frameworks (AWT and Swing) are not truly model-based in the way defined by the original and successor Smalltalk GUI frameworks. In that original conception, the model of the MVC framework is a ValueModel as described above: a notifying holder of a value. This ValueModel concept could be used in many different areas of a program and so provided a very flexible interface between subsystems. This flexibility was especially used in the loose coupling of the View-Controller (or Widget if the responsibilities were combined) to the Model that the View-Controller presented and altered\*. Essentially the entire behavior of the UI could be described by Models connected to other Models, and the View-Controllers (whether Widgets, Panels, or Windows) were simply attached to the Models to produce the Graphical interface.

Although they had been approaching an MVC-like architecture, Java’s GUI frameworks still did not have this general model-based capability†. So we needed to augment Java’s GUI frameworks to understand it. To augment an OO framework with new general behavior, a developer has three options:

1. Provide framework clients that are adapters to the new behavior

\* For an explanation from the original source of the MVC concept, see [Reenskaug+WL 98]. For some of the enhancements of MVC within Smalltalk over the years and other perspectives, see [Burbeck-1], [Krasner+P 88], [Leibs+R 92], [Lunt-1], [Woolf-1], and [Buschman+MRSS 96]

† Swing has “models” but most of these are very GUI-focused and are really semi-graphical UI elements more than a true model as originally conceived.

2. Subclass framework classes at acceptable extension points
3. Change the framework code itself

Since Swing is a standard, Javasoft-provided library, the third option was not acceptable. Besides making our development efforts more difficult, it would almost certainly prevent us from using an IDE and this would be a serious penalty in trying to make development easier on application developers. Among the first two options, we needed to use both approaches depending on how Swing handled extension and its relationship with the Swing clients. Because of the particulars of Swing, we tried to use the adapting-client approach as much as possible and to have little or no code changes in extension classes.

### ***Integration with the GUI builder***

To support the Java GUI builder\* required supporting a textual configuration mode somewhat similar to the VisualWorks UI layout mode: there would be a stage where objects had not yet been hooked up to other objects but instead these objects had a textual key that would enable them to find and connect to their partner. By having this ability, an application developer could simply type in the required name/key into the bean property editor and the framework code would do the rest of the hookup. These keys could be standard OQL/OCL path expressions (“person.name.first”) to support navigating through the properties of the domain objects.

### ***Difficulties with “final” and “private”***

One reoccurring difficulty with moving to the Java environment was the prevalence of extension-restricted classes and methods in the available libraries. Frequently classes and methods were restricted with “final” or “private” keywords that made the class unusable for building new functionality. Instead, whole classes would have to be reproduced to allow their functionality to be tailored to our specific needs. A simple example of this was PropertyChangeSupport, a helper class that allows an object to delegate most of the implementation required to support JavaBean property change notification. Because of several “private” restrictions, the existing PropertyChangeSupport could not be made to work with the ValueModel concept and we needed to create a new VMPropertyChangeSupport class that was unrelated to PropertyChangeSupport by inheritance. In other cases (like Vector) this caused problems with inter-system interfacing because clients were typed to a Class (that could not be extended) instead of an Interface.

### **Customizing to Island Pacific needs**

Island Pacific had a few goals that were not covered by the original migrated frameworks. The most significant goal was supporting a multi-threaded UI-to-Domain interaction where the success and results of a Domain operation may return asynchronously relative to the request from the UI. For example, the user could be entering information for a new employee. If the entered ID number conflicted with an ID currently within the system, the field for the ID number would indicate an error. But to support the rapid entry of information, the check upon leaving the ID field would not “block” the user from continuing to enter a name, address, and so on. The check of the ID field would be done asynchronously and the results posted to the field if there was a problem.

To support this and similar asynchronous notification, a ProblemRecorder was sent with all modification operations so instead of:

```
public void setValue(Object newValue);
```

we would have

```
public void setValue( Object newValue, ProblemRecorder recorder );
```

If there were any business rule or simple type errors, the recorder would be told and that would trigger back notification to the UI layer itself even if the main GUI thread had moved on to other activities.

---

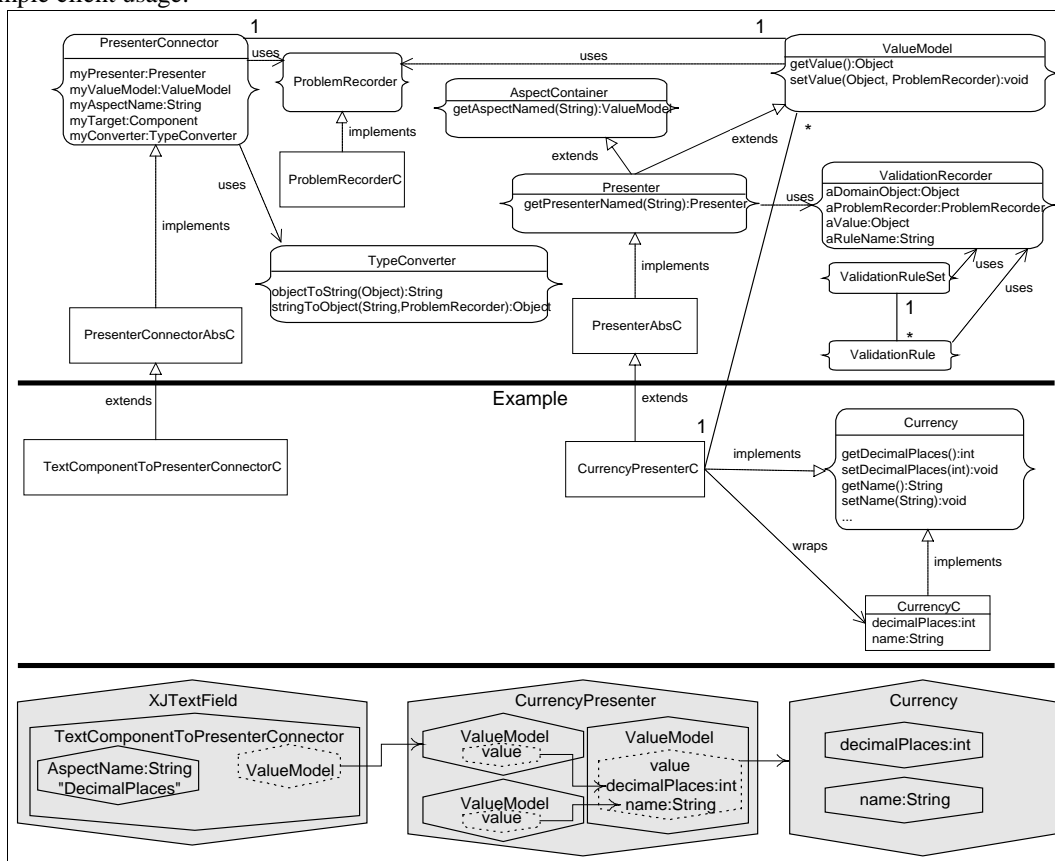
\* In our case, the JavaBean-based VisualAge GUI builder, but we were now using just standard component editing and not the visual programming language drawing capabilities.

# The Resulting Framework

The final Island Pacific Domain–Presentation framework consisted of nine core concepts:

1. DomainObjects
2. ValueModels
3. Aspects and AspectContainers
4. Presenters
5. PresenterConnectors
6. TypeConverters
7. ProblemRecorders
8. ValidationRules
9. GUI Widgets and GUI Panels

The following diagram shows the static relationships among most of these framework concepts and an example client usage.



All the Types shown above are represented in Java as interfaces. As will be discussed later, some of the client classes can be automatically generated given a DomainObject.

## Description of Concepts

Each of the following sections describes a concept from the framework.

### Domain Object

A **DomainObject** captures knowledge about a domain (a representation of a business within a computer). DomainObjects manage the behavior, data, and rules that apply to the business and usually live within a transactional environment. In our framework, DomainObjects have no responsibilities or capabilities to notify others of changes. These responsibilities are instead given to the DomainPresenter so that the

DomainObject can focus on just its inherent business behavior and not be trapped within the notification capabilities.

## Value Model

The **ValueModel**, as described before, is an abstraction that is used heavily in this framework. The value model is a simple structure that keeps track of one property, its value. This simplicity allows us to use them as wrappers around each property of a domain object. Wrapping a property in such a way successfully masks its inner complexities and allows us to treat all properties of a domain object in the same exact manner, thus enabling us to pass the value between layers the same way. The value model is a powerful construct, but much of its power comes from its co-existence with an Aspect Container.

## Aspect and Aspect Container

An **Aspect** is a property of an object represented as a ValueModel. An **AspectContainer** is any object that can provide Aspects for its properties. Typically, an aspect container is responsible for providing access to the value models wrapping the properties of a single domain object. Because a value model can wrap a distinct property and it can also wrap a value that is derivable from 1 or more distinct properties of a domain object, it is helpful to think of this collection of wrapped values as a collection of different aspects. It becomes possible to identify each of these value models/aspects by a name held in a String object. When this occurs, we find that regardless of whether an aspect is physical or derived, we can obtain a generic value model reference to it by simply asking the aspect container for it by name. This becomes very important in the GUI layer when we wish to tie a widget to an aspect of a domain object.

## Presenter

At the core of the presentation layer is the Presenter. The **Presenter**, as its name suggests, is responsible for presenting information from the domain layer. Because the presentation layer is free of dependencies on any one particular form of presentation, it is suitable as a link between any UI, be it graphical, textual, or other. The presenter inherits its behavior directly from the value model and the aspect container. With such heritage, we know that the presenter must be able to hold onto a value, and also provide references to value models of aspects of a domain object when asked for them by name. The presenter uses these two behaviors in unison to provide glue between the domain and GUI layers in our framework. Presenters are created for each type of interesting domain object. This is done so that the presenter can have a closed set of aspects from which it is expected to be able to deliver. In order to retrieve these aspects, the presenter must have access to a domain object that contains data layer information. This is where the value model heritage comes in, the presenter is able to hold onto the value of one domain object at a time. Using this held value, the presenter serves up aspects/value models that are themselves wrappers around the presenter's value model value. Since the served up aspects are bound to the presenter's value model, we are able to swap out the domain object at will and inform any these aspects (value models) that they should update themselves and any other object that cares about their value based on the new domain object.

## Presenter Connector

Gluing the presentation layer to the GUI is the job of the **PresenterConnector**. A presenter connector must have references to a GUI widget and a presenter in order for it to do its job. Once it has these references, it need only be told what aspect the widget is interested in and it will ask the presenter for the appropriate value model. Having done this, the presenter connector begins listening to change events on the value model retrieved from the presenter. Depending upon the type of GUI widget it is attached to, it will also begin listening in on the appropriate change event of the widget. The purpose of listening in on both sides is to enable the automatic update between the two layers. If we take, for example, a textfield, the presenter connector would listen in on the textfield's LostFocus event to know when it should take the textfield's String value and propagate it to the presentation layer for forwarding to the domain layer. Likewise, the presenter listens for the value model's property change event to know when to propagate the value model's new value up to the textfield. Now if we are listening in on an aspect of, say, a Currency object that represents the number of decimal places, we know that the value is going to be some form of integer, either

a primitive int or an Integer object. This immediately presents a problem because the textfield can only display and receive input in the form of a String. In order to handle this dilemma, the presenter connector is fitted with a Type Converter.

## Type Converter

The **TypeConverter** is an interface that understands two methods:

```
public String objectToString( Object anObject );
public Object stringToObject( String aString, ProblemRecorder aProblemRecorder );
```

Since a value model can only hold an Object, we know that we do not have to deal with primitives at this level, in fact, it is at the presenter/value model intersection that primitives are converted into their appropriate Object wrapper type. For each type of domain object that might need to be converted between string and object, a converter should be implemented. An example where this might occur is in an Employee object. It could be desired that a textfield represent an employee's name. When a name is entered, the underlying domain object that the textfield is attached to would be required to change appropriately. In this case a TypeConverter with a custom stringToObject() method would be required for this transaction to complete successfully.

Knowing that a GUI can only deal with objects ( in general ) as Strings for display or input, this interface captures all of our conversion needs at the presenter connector level. We know that in general, we can always obtain a String representation from an object without error, but obtaining an object from a String is not always as easy. To handle this possibility of failure, the stringToObject method requires that a ProblemRecorder be passed into it for the recording of errors.

## Problem Recorder

The **ProblemRecorder** is a passive construct that we pass along into critical methods that could normally throw exceptions. We decided upon this construct for two main reasons. First, we found that throwing exceptions to indicate errors made for an unstable application and made initializing domain objects difficult at times when interdependencies existed. Secondly, it is our current belief that we should only throw exceptions when there is a critical problem that should cause the application to come to an end.

## Validation Rule

In order to localize all of the business domain logic, we utilize a trio of classes. At the lowest level is the ValidationRule. The **ValidationRule** contains the code that satisfies a single business rule. A rule can be used to validate a single value, a relationship between multiple values, or the state of a domain object as a whole. The Validation Rule Set is a logical collection of rules, typically all of the rules needed by a single domain object. A Validation Rule Set allows a rule to be validated by name. Both a single Validation Rule, and a Validation Rule Set require a single Validation Recorder in order to perform the validation. A Validation Recorder maintains properties for the name of the rule, the proposed new value, the domain object (interface) containing the property to be set, a problem recorder, a reference to the previous value, and a flag for indicating success or failure.

## GUI Widget and Panel

In order to pull this framework together, it was more effective to adapt the current GUI framework (Swing) to our needs rather than try to build one that met our needs. The adaptation process was relatively simple. We decided that since the presenter connector needed a reference to the GUI widget it was connecting, it was convenient to embed the connector within each widget. By doing this, we ensured that each widget by default had the correct type of connector, and also removed one development step each time a widget was used. However, since the behavior of the connector is very discreet, we didn't want to burden the developer with knowing about the connector. To handle this, we had each of our adapted widgets extend an interface that acted as a façade to the connector. The façade simply requires that the developer specify the name of

the aspect the widget should reflect and the type of converter ( if any ) that was needed for that aspect. The only thing left over is specifying the presenter so that the connector can make the two way connection between the GUI and presentation layers. This was automated by adding intelligence to an extended version of JPanel that contains a presenter reference that automatically gets added to widgets as they are added to the panel.

## *Descriptive Walkthrough*

The resulting framework begins with the domain layer described in terms of interfaces. The domain is then implemented in a class that implements the interface that acts mainly as a storage device for information retrieved from or to be sent to the data layer. In the domain presentation layer, a presenter is created for each interesting domain interface by wrapping the concrete class implementation of the interface. The wrapping is achieved by using a value model for each concrete or derived value of the domain implementation. Since the value model only understands a generic `setValue( Object anObject )` and a generic `Object getValue()` method calls, the presenter and the value models work together to call the correct setter and getter methods of the wrapped domain object. In addition to these four constructs, the business domain rules have been encapsulated into individual rules that know how to validate business requirements. Individual rules are brought together to form rule sets that typically coincide with domain objects (interfaces). These rule sets understand how to accept a domain interface, a value, and a rule to perform to check for business validity. In order to support the capturing of problems with the business validation, and not wanting to throw exceptions for these types of problems, we created the Problem Recorder. The Problem Recorder is your basic visitor pattern that is sent along with a request and records any interesting anomalies. A problem recorder is passed into the rule set validation routine and then examined by the caller when the validation returns. In our case, the domain presenter passes it in and examines it upon return. Typically, the problem recorder is passed into the presenter from a higher level in the framework, the connector.

As mentioned before, the connector is the intelligent glue that controls the interactions between the GUI widget and the domain presenter / domain. The connector is responsible for listening to change notifications from the domain presenter as well as listening for change notification from the GUI widget so that these modifications can be propagated between layers. An important responsibility of propagation is the conversion between the representations needed at each layer. In order for these interactions to occur, the connector needs to know three things, the domain presenter it should talk to, the aspect it should ask the domain presenter for, and the GUI widget it should interact with. The GUI widget requirement is taken care of by embedding the connector into the GUI widget itself. The other two pieces are simply String references to the real instances. Real instances of the value object are obtained by asking the domain presenter for the value by name. The domain presenter instance is obtained at runtime by asking for it by name from its containment parent. Domain presenters are automatically propagated down the containment hierarchy by subclassing the JPanel class to be knowledgeable of connectable widgets and subclassed JPanels.

Extending the JPanel widget meant overriding the "add" methods to be aware of widgets that contained connectors. Adding this intelligence allowed us to make a panel the holder of a domain presenter. These two added features allowed us to automatically set the domain presenter of "connectable" widgets as they were added to the panel. Panels exhibit similar behavior in that as one panel is added to another panel, the added panel inherits the domain presenter of its new parent. In addition, panels can also specify an aspect string that will change its domain presenter, to a subordinate domain presenter of its parent presenter. With these features, the GUI designer needs to know only which aspects a domain presenter supports and then add panels and widgets that refer to the correct aspect names. In the end, when all of the panels are placed into other panels, they are ultimately placed within a dialog, a frame, or an applet. At this level, the developer needs only set the master application domain presenter and domain object once at this top level and then at runtime, these values will ripple all the way to the lowest level of panel detail. As these values ripple their way down, each panel and widget connector sets itself up to reflect and update domain values automatically. All the GUI developer has to do is lay a few widgets out in the visual editor, type a few Strings in that represent the appropriate aspect to reflect, and compile.

The end result is that the GUI developer only needs to concentrate on "drawing" the application and making "notations" on each widget as to what property it represents. Once the developer is done drawing and notating, the application "automatically" connects itself to all the proper domain objects, checks business rules automatically, and displays problems with rule validation.

## *Issues Encountered*

The original domain presenter, which is a derivative of the aspect adapter, only supported retrieval of value models that held the values of the aspects we were interested in. This meant that in order for two different panels to retrieve an aspect from the same type of domain presenter, each panel had a unique instance of that domain presenter. This resulted in updates from one panel not getting reflected in the other panel because even though they were both looking at the same domain object via a domain presenter, they were not looking at the same domain presenter. In this model, it is the domain presenter that is responsible for change notification. To remedy this, we modified the domain presenter to be responsible for handing out domain presenters for each type of value it could hand out. Doing this allowed us to not have to specify an actual instance in each panel, but an aspect name that referred to an actual instance in the master domain presenter's hierarchy, thus ensuring that all panels interested in a particular type of domain presenter all referred to the same instance. It was this addition that also allowed us to be able to automatically link domain presenters to the connectors contained in the GUI widgets and ripple these settings through the application containment hierarchy.

## *Tradeoffs, Improvements, and Benefits*

The main tradeoff from this third phase of our development from our second phase is that extracting the change notification from the domain object resulted in twice as many classes. The resulting architecture requires two classes for each business entity: one for the domain layer and one for the domain presentation layer. However, because each objects' responsibilities are so specific, all domain classes share an identical form and all presentation classes share an identical form. As a result of this similarity, we were able to successfully create code generation tools that are an integrated piece of the VAJ IDE. These code generation tools allow us to define only the domain layer interfaces and then auto-generate the domain implementation class and domain presentation class. In most cases, this is a time savings ratio of 5minutes of generation time to 2 hours of hand coding time per domain class. This, in addition to the structure and ease of GUI building resulting from this new framework, more than makes up for this tradeoff.

The greatest benefits of this new framework are apparent in the overall amount of application specific code that is needed for a GUI. The overall number of lines of code between our phase two implementation of an application and the phase three implementation of the same application is relatively the same. However, due to the nature of the application suite we are to build, the domain layer and presentation layers are shared among applications so the lines of code will differ exponentially as we build the rest of the GUIs in the application suite. This is mostly due to the fact that the new framework centralizes all the business logic and domain needs such that the GUI is free of this code, making it very lightweight. Our phase two implementation on the other hand, had very heavyweight GUIs that re-implemented business logic in multiple areas, making domain knowledge non-reusable and decentralized.

## **Conclusions**

---

Leveraging the quality of an existing framework paid off for Island Pacific Systems Corporation. In building The Eye™, the decision to migrate the Domain Presentation framework into Java made our application better behaved, easier to code, and easier to debug. The resulting framework was better for our needs than our initial attempts at meeting these needs from first principles, and it was better than the approach of extensive use of a Visual IDE. Although the concept of a clean UI–Domain layering is simple to understand, an existing framework (even one written in a different programming language) fills in many important details.

The following are some suggestions to projects considering leveraging existing frameworks:

1. **Look at the available work for several languages in the areas you are interested in.** A tremendous quantity of excellent work in OO design and patterns was created during the last twenty years, but most of it will be in languages other than your current development language. The value of the work usually far exceeds the cost in translating it.
2. **Focus on the specific needs of your project and the target environment.** It can be tempting to try to migrate an entire, very general, framework to another language but there may be a significant mismatch that either prevents or makes very expensive such a complete migration. A partial, project-specific migration will be much more effective.
3. **Try to keep core concepts.** The biggest advantage of migrating an existing framework is to keep the subtle interrelationships that made the framework successful but might be hard to come up with on your own. Although much of the framework may need to be “adjusted”, try to keep the core concepts within it.
4. **Consider whether any commercial products meet your needs.** It is much more productive to use an available product than to take a framework, migrate it, and maintain it. The migration approach is only cost effective if your needs are significantly different from available products.

Sometimes it seems that the software industry loves novelty: portions of the industry migrate from one language to another just for fun or fashion. But in other cases, a new language has some real advantages and the important issue is how to reap the value of both these new capabilities and all the previous investments. The ability of a language to leverage existing work and ideas may become the prime determinate of the language’s success. There were many annoyances with moving to Java (immaturity of its libraries, the use of “final” and “private”, a relatively weak type system), but overall it did support the migration of a good framework from a significantly different OO language. And this migration increased the ability of the Island Pacific development team to provide solutions.

## References

---

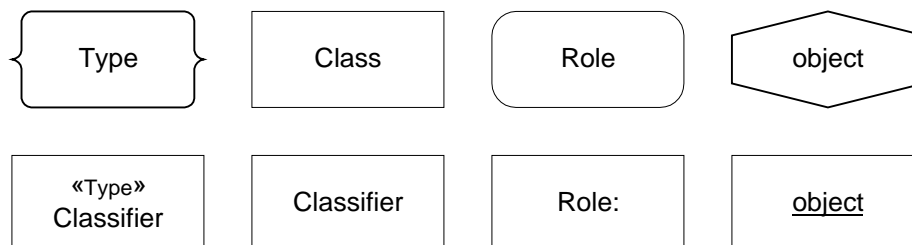
- Bergin+G 96** Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., editors. *History of Programming Languages – II*. Addison-Wesley, Reading, MA, 1996.
- Burbeck-1** Steve Burbeck, Ph.D. “Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)”.  
<http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html>
- Buschmann+MRSS 96** Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns: Pattern-Oriented Software Architecture*. Wiley, Chichester, England, 1996
- ChiMu-1** ChiMu Corporation. “Kernel Frameworks”  
<http://www.chimu.com/publications/kernelFrameworks/index.html>
- ChiMu-2** ChiMu Corporation. “ChiMu OO and Java Development: Guidelines and Resources”  
<http://www.chimu.com/publications/javaStandards/index.html>
- Coad+M 96** Peter Coad and Mark Mayfield. *Java Design: Building Better Apps & Applets*. Yourdon Press, Upper Saddle River, NJ, 1996.
- Coplien+S 95** James Coplien and Douglas Schmidt, Editors. *Pattern Languages of Program Design*. Addison-Wesley, Reading, MA, 1995.
- Eckstein+LW 98** Robert Eckstein, Marc Loy, and Dave Wood. *Java™ Swing*. O’Reilly & Associates, Sebastopol, CA. 1998.
- Firesmith+E 95** Donald Firesmith, Edward Eykholt. *Dictionary of Object Technology: The*

*Definitive Desk Reference*. SIGS Books, Inc., New York, NY, 1995.

- Fussell-1** Mark L. Fussell. "SmallJava: Using Language Transformation to Show Language Differences"  
<http://www.chimu.com/publications/smallJava/index.html>
- Gamma+HJV 95** Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Object-Oriented Architecture*. Addison-Wesley, Reading, MA, 1995.
- Goldberg+R 83** Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.
- Gosling+JS 96** James Gosling, Bill Joy, Guy Steele. *The Java™ Language Specification*. Addison-Wesley, Reading, MA, 1996.
- Howard 95** Tim Howard. *The Smalltalk Developer's Guide to VisualWorks*. SIGS Books, New York, NY, 1995.
- Kay 96** Alan Kay. "The Early History of Smalltalk" in [**Bergin+G 96**].
- Krasner+P 88** Glenn. Krasner and Stephen. Pope. "A Cookbook for using the Model-View-Controller User Interface in Smalltalk-80," *Journal of Object-Oriented Programming*, Vol. 1 No. 3, August/September 1988, pp. 26-49.
- Liebs+R 92** David J. Liebs and Kenneth S. Rubin. "Reimplementing Model-View-Controller" *Smalltalk Report*, Vol. 1, No. 6, Mar./Apr. 1992, pp. 1-7.
- Lunt-1** Eric Lunt. *MVC Widgets Tutorial*.  
<http://csis.pace.edu/~bergin/Java/elunt/Doc/index.html>
- Martin+RB 98** Robert Martin, Dirk Riehle, and Frank Buschmann eds. *Pattern Languages of Program Design 3*. Addison-Wesley, Reading, MA, 1998.
- Reenskaug+WL 96** Trygve Reenskaug with Per Wold and Odd Arild Lehne. *Working with Objects: The Ooram Software Engineering Method*. Manning, Greenwich, CT, 1996.
- Vlissides+CK 96** John Vlissides, James Coplien, and Norman Kerth, Editors. *Pattern Languages of Program Design 2*. Addison-Wesley, Reading, MA, 1996.
- Woolf-1** Bobby Woolf. "Understanding and Using ValueModels" in [**Coplien+S 95**].  
<http://home.att.net/~bwoolf/ValueModels/vmodels.html>

## Notation

The notation used within this document is UML with the following enhancements/stereotypes. To provide a visual clue of the differences between a Type, a Class, a Role, and an Object, the following icons are used:



Their meaning is exactly the same as for UML.