# XForms
# An Interactive Forms Design Tool

**Abstract**

*For this project, I designed and implemented XForms, a versatile, Mac OS X forms design tool that automatically generates forms in multiple media formats from a single design source. XForms can generate both traditional paper-based forms and web-based HTML forms. In addition, a form design can be output as an XML or PDF document.*

*I wrote XForms in Objective-C using Apple's Cocoa framework as a proof-of-concept application designed to be easily extendable to support any number of form output formats.*

**December 1, 2005**

**Faculty Advisor**
Dr. Raphael Finkel

**Masters Candidate**
Nolan W. Whitaker

# Table of Contents

Summary ........ 3
Introduction ........ 3
  My Background ........ 3
  Paper Forms ........ 4
  Electronic Forms ........ 4
    Legacy Forms ........ 4
    Data Storage and Retrieval ........ 5
    Routing and Processing ........ 6
    Signatures & Authentication ........ 6
XForms ........ 7
  Usage ........ 8
  Implementation ........ 9
  Model View Controller Design Pattern ........ 10
    Model ........ 11
    View ........ 12
    Controller ........ 13
    MVC Class Interactions ........ 14
    Form Output Options ........ 15
  Future Extensions and Modifications ........ 15
Appendix: Detailed Class Diagrams ........ 16

## Summary

For this project, I designed and implemented XForms, a versatile, Mac OS X forms design tool that automatically generates forms in multiple media formats from a single design source. XForms can generate both traditional paper-based forms and web-based HTML forms. In addition, a form design can be output as an XML or PDF document.

I wrote XForms in Objective-C using Apple's Cocoa framework as a proof-of-concept application designed to be easily extendable to support any number of form output formats.

## Introduction

When personal computers (PCs) first began to appear on office desks, a plethora of articles by information technology professionals predicted a dire future for paper books and forms. Despite those promises of a paperless office over the past two decades, paper-based forms comprise an even larger part of every major commercial and public sector enterprise in existence today, even though low cost computing hardware is found in most every office[1]. In fact, I believe this increase in the quantity of paper-based forms is due to the personal computer.

### *My Background*

I am a computer science masters candidate at the University of Kentucky with nearly seven years of experience with the second-largest information-technology integration and consulting company in the world, EDS. While there I helped design, develop and integrate a number of software packages for the U.S. Army Recruiting Command (USAREC), some of which were deployed to over 14,000 users nationwide. Chief among these products is an application to automate collecting, entering, storing, and routing over 100 enlistment forms; this application significantly increased Army recruiters' productivity, cutting the time needed to collect information and fill out enlistment forms from recruits.[2] After fielding the application, the Army met its recruiting goal for the first time in many years.

[1] Sellen , Abigail J. and Harper, Richard H. R., *The Myth of the Paperless Office,* MIT Press, 2001.
[2] "Army Recruiting Information Support System, Improved Readiness Through Enhanced Recruiting." EDS sales promotional literature. Copyright 1999, EDS. http://www.usarec.army.mil/ariss

## *Paper Forms*

In my observation of industry and government, rather than eliminating paper-based forms, the PC has become a catalyst for the proliferation of forms used by enterprises today. Word processors in the 1970s, desktop publishing in the 1980s, and specialized forms-development software in the 1990s each made it easier for the average office bureaucrat to create a unique set of forms, giving rise to a myriad of problems caused by nonstandard and poorly designed forms.

Before the advent of the personal computer, forms had to be sent out to be typeset and then printed by a professional printing organization. With the advent of the word processor, businesses could generate their own forms. Many did just that. In 1984, the Macintosh computer opened up the world of the professional publisher to everyday office users, allowing everyone to create nearly professional, typeset-quality documents and forms.

In the 1990s, specialized forms design products entered the marketplace. At first, these products were essentially page-layout products not much unlike the desktop publishing software that came out in the 1980s, except that they were specifically designed to make forms layout, design and management easier. As these packages matured, they began to provide support for electronic-based forms, creating the first opportunity for a business to truly become paperless—at least in theory.

## *Electronic Forms*

The American Heritage Dictionary of the English Language defines a form as "a document with blanks for the insertion of details or information.[3]" An electronic form, or **e-form**, is document in a computer-readable format that allows a user to enter information. The e-form may encapsulate user data within the form itself, or the data may be stored independently.

Once an organization decides to migrate toward integration of e-forms into its business processes, the organization needs to consider a number of issues. I outline several below and then show how XForms fits into the overall picture.

### Legacy Forms

One of the issues facing any organization contemplating the switch to e-forms is the presence of legacy paper forms. There are essentially four approaches to the move to a paperless office.

The first and most easily adopted solution is to move all future forms to an electronic version, leaving all legacy forms non-automated and handled in the current fashion. The

[3] *The American Heritage® Dictionary of the English Language,* Fourth Edition.

advantage of this solution is that it is least costly. However, it makes data search and retrieval very time consuming because one must check two locations for all customer documents, the electronic and paper archives.

A second solution is to scan all legacy documents, storing all printed documents in a large image database. The advantage of this solution is that imaged documents can be linked to current database records and retrieved electronically. However, this solution poses a number of problems. Often these forms have deteriorated due to time, handling by office workers, shuffling from one storage location to another, and physical damage from exposure to the elements. Scanning the documents and getting a usable image can be difficult. Often this solution falls far short of user expectations in that, although the documents are in a database, document contents are generally not searchable because they consist only of images. Thus, unless the associated database record is known (for instance, the customer record), finding related images is next to impossible. However, the same problem exists with paper-based documents, since they can only be located based upon their filing index.

A third solution is a modification of the previous solution. Again, in this solution, one scans legacy documents store them in an image database, but in addition, one stores text-searchable versions of each document as well. Obviously, this solution is far more resource intensive and requires substantial data entry, with significant risk of transcription error due not only to the normal error rate associated with data entry tasks but also due to poor readability of older documents.

A fourth solution is to use a hybrid of the above techniques to ease transition to a paperless office. This is the solution we created for the US Army Recruiting Command. This type of solution moves the organization to a paperless office with the ability to create paper-based copies of each form when forms used by the office must be forwarded to outside groups not participating or compatible with electronic solutions. For outside enterprises willing to use electronic versions of the organization's forms, there are various schemes in the marketplace to allow organizational partners to share electronic access.[4]

## Data Storage and Retrieval

Once it has selected a broad electronic-forms strategy, a company must address the issue of data storage and retrieval, in particular, how to efficiently store documents. Forms are generally composed of several **elements**. These elements may include blocks of boiler-plate text, images, line-art and data-collection areas.

One option is to store form elements and form data together. However, this approach is rather inefficient because the data contained on a form generally requires only a small

[4] I am under a non-disclosure agreement and cannot discuss specifics of the EDS Army solution. EDS created a marketing piece describing the solution referenced in note 2 above and demonstrated part of the laptop-based forms automation software and video presentation at a University of Kentucky recruiting fair. I can provide a copy of the EDS marketing piece to anyone desiring a copy.

percentage of the storage space needed to store the form elements.[5] The obvious solution is to split data from form elements and store each separately. However, this approach presents a number of problems.

The chief problem encountered when separating form elements from form data is form versioning. Invariably, a single form will undergo numerous revisions in its lifetime. Electronic forms design tools encourage this behavior because they enable staff to make and distribute changes electronically. It's imperative that each form version be archived and provision made at the start to relate entity data with the form version with which it is associated. Failure to do so can be disastrous.

## Routing and Processing

Creating and storing forms in electronic format are only two of the challenges of going paperless. Often one individual creates a document that a number of individuals must read and agree on before the organization can accept it. This process and collection of approvers is known as an approval chain.

**Approval chains** are rule-based and range from very simple, linear rule sets with one or more approvers to rule sets comprised of a complex graph of rules, approvers, forms, and approval levels. With more complex rule sets, a forms-routing solution must move the document to one or more approvers based upon the response or non-response of an approver within a specified criterion (e.g.: time period) or collection of criteria. Document routing, in and of itself, is a rather complex process and is often ignored by many organizations resulting in a mess— printing paper copies of documents and routing them using legacy business practices and processes, leaving the organization disappointed with its automated solution.

### Signatures & Authentication

While a document is in route, electronic forms software may need to allow a form's data to be altered or changed. Sometimes alteration invalidates the document; other times, supporting alteration is a requirement of the forms-routing solution. In any event, for a document (whether it be a contract, credit application, health form, or other document) to be enforceable in court, there must be a way to show that it has not been materially altered since the person with whom it is to be enforced against signed off on its contents[6]. If a document is materially altered, then any prior signoffs must be invalidated if the document is to be enforced against that person. However, there are instances when a document will never be enforced in court or where information is to be added by one or more person at some point in the approval chain. For example, a medical insurance analyst may be required to add comments to a document justifying a test before an expense is approved; this type of document modification occurs after it is signed by both the patient and the primary-care physician but should not invalidate the signature of

[5] This claim is based upon personal, professional experience and observation.
[6]See the Millienum Digital Commerce Act of 2000, Public Law No: 106-229.

either patient or physician. The approval chain rule set must address each of these possibilities and many more for every form.

With paper documents, holographic signatures are used to signify that a party agrees to the information contained on the document. Documents with holographic signatures suffer from three serious problems: signer authentication, document integrity, and repudiation. Authentication refers to issues surrounding the identification of the parties with whom a form is related. Integrity relates to the issue of whether a document has been materially altered between the time it was signed and some date in the future. Repudiation refers to the problem of a person denying signing the document.

Despite these problems, holographic signatures are used on most every legal instrument in existence today. In some cases, a trusted third party, a Notary Public, is employed to mitigate the authentication and repudiation problems.

e-forms are no different from paper forms in that regard; they have the same three potential problems: authentication, integrity, and repudiation. There are a number of ways beyond the scope of this paper to address each problem using a **digital signature**. A digital signature is an assurance, using cryptography, of authentication, integrity, and non-repudiation.[7] It may or may not include an image of a person's holographic signature.

Essentially, there must be a way to digitally sign each document, a change log showing any document alterations, and a set of rules to determine what constitutes material alteration and to invalidated selected form elements when material alteration occurs. This is a very complex subject and is not addressed by XForms.

## XForms

I designed XForms, a Macintosh OS X application, to address only a small portion of the electronic-forms automation problem domain. In particular, my work is restricted to the design and creation of web and paper-based forms. I do not address legacy forms, data storage and retrieval, routing and processing, or signature and authentication issues.

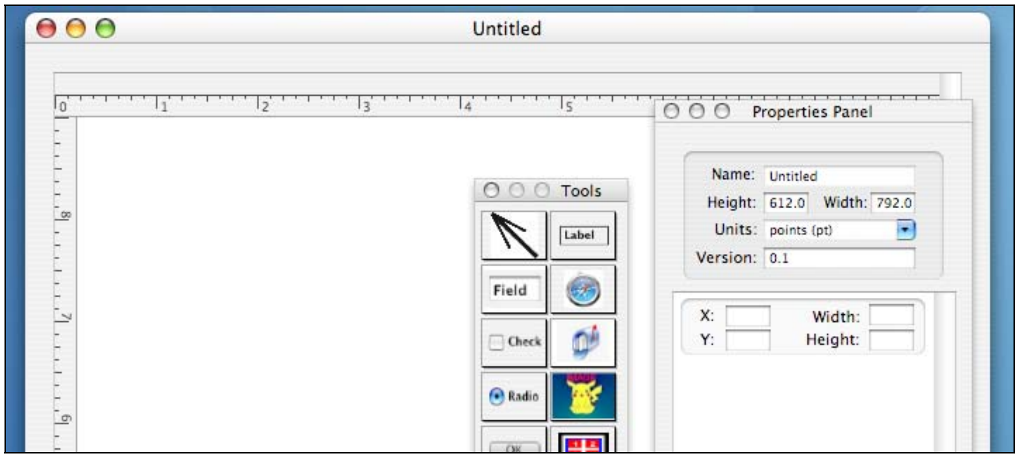
In undertaking this project, I had four objectives: learn objective-C, learn the Cocoa development framework, create a pure implementation of the model-view-controller design pattern as presented in the text *Design Patterns*[8], and create a useful application that generates both paper and web forms from a single design source.

[7] Mel, H.X. and Baker, Doris, *Cryptography Decrypted*, Second Printing March 2001, p. 115.
[8] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*, 30th printing, September 2004.

## Usage

To start XForms, double click its icon. Once the application launches, you see a blank form, a pallet of form elements and a properties panel similar to that pictured below.

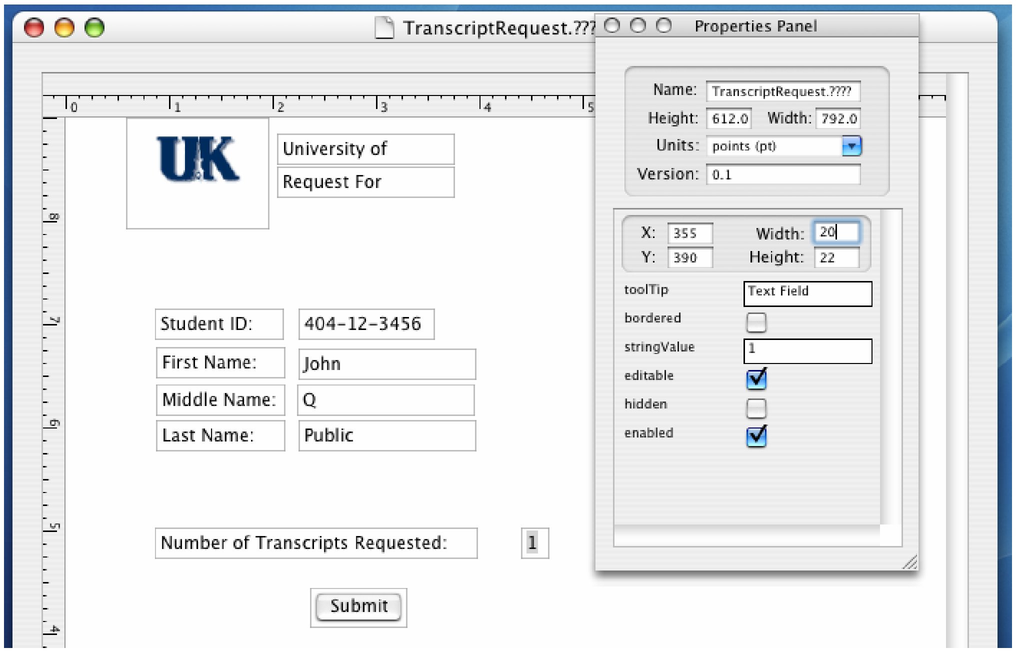
**Figure 1: Form design startup**



Name: Untitled
Height: 612.0 Width: 792.0
Units: points (pt)
Version: 0.1
X: Width:
Y: Height:

To add an element to a form page, drag it from the Tools pallet onto the page. Once you do that, the properties panel dynamically generates a list of properties that you can change for the selected element.

**Figure 2: Properties panel**



University of
Request For

| | |
|---|---|
| Student ID: | 404-12-3456 |
| First Name: | John |
| Middle Name: | Q |
| Last Name: | Public |
| Number of Transcripts Requested: | 1 |

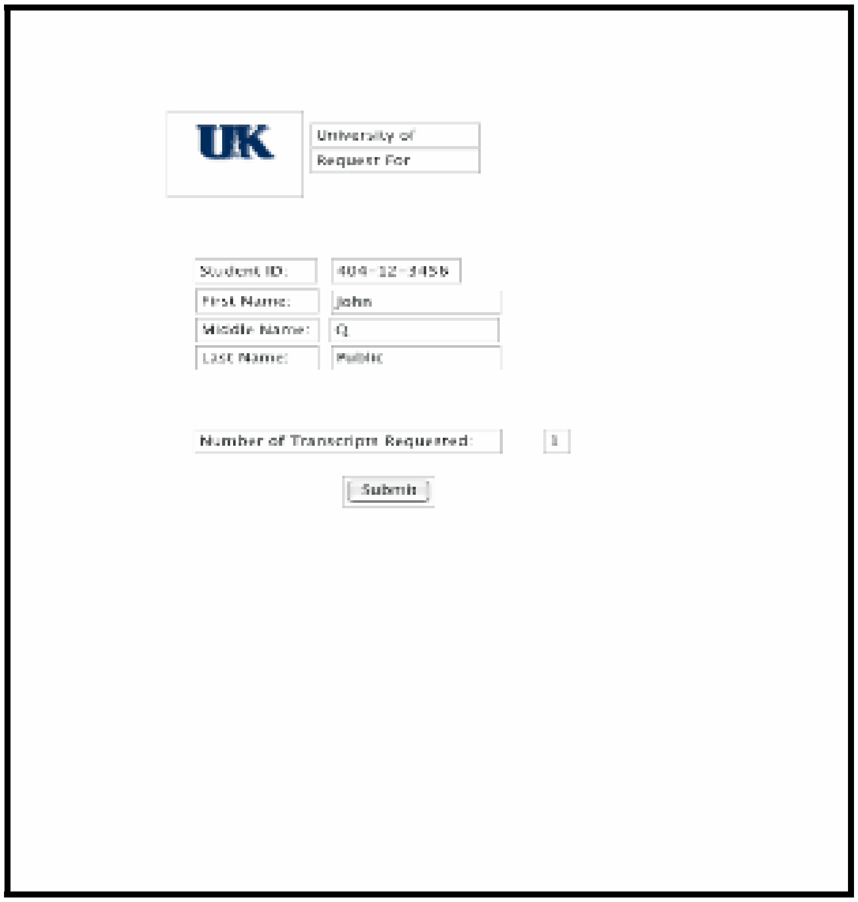
Name: TranscriptRequest.????
Height: 612.0 Width: 792.0
Units: points (pt)
Version: 0.1
X: 355 Width: 20
Y: 390 Height: 22

| | |
|---|---|
| toolTip | Text Field |
| bordered | ☐ |
| stringValue | 1 |
| editable | ✓ |
| hidden | ☐ |
| enabled | ✓ |

The properties panel allows you to control placement, size, and properties specific to each form element. For example, an image element allows you to specifiy a graphic image filename; whereas a checkbox element does not. XForms maintains property lists that a form designer can change and dynamically generates the property panel view based upon the list corresponding to the selected element on the page view. XForms outlines selected elements in blue in the page-design view. If you change a property, XForms propagates it to the element in the form view immediately.

Once you have designed a form, XForms can be print or save it in one of four formats: a proprietary binary format, XML, HTML, or PDF. Choosing either XML or HTML allows you to manipulate the form design in a third-party utility—a web design tool or a forms viewer, for example. Binary and PDF formats encapsulate all images in the form. XForms' PDF format is read-only and does not support Adobe's form fill-in features. To output a paper form, select the option to print from the main menu.

**Figure 3: Print Output**



UK

University of

Request For

| | |
|---|---|
| Student ID: | 404-12-3456 |
| First Name: | John |
| Middle Name: | Q |
| Last Name: | Public |

| | |
|---|---|
| Number of Transcripts Requested: | 1 |

## *Implementation*

I wrote XForms in Objective-C using the Cocoa framework, a version of the OpenStep framework created by the engineers at NeXT Computer in the late 1980s and mid 1990s[9]

[9] Garfinkel, "Steve Jobs and the History of Cocoa, Part One," O'Reilly Mac Dev Central, http://www.macdevcenter.com/pub/a/mac/2002/05/03/cocoa_history_one.html.

to support a Rapid Application Development (RAD) paradigm. OpenStep was one of the first reasonably successful commercial ventures offering a RAD platform that consists of an integrated development environment with editor, visual user interface designer known as Interface Builder, and code framework. With Apple's extensions, Cocoa has become a very complex, feature-rich, application-development framework. There are few books documenting the framework. Consequently, it is very difficult to learn in a short span of time. There is an open-source implementation of OpenStep known as GNUStep[10].

Objective-C is an object-oriented extension to the C programming language that remains fully compliant with the ANSI standard for C. Objective-C is not officially documented by a world standards body. Apple's version of the Gnu compiler, GCC, is used to compile Cocoa applications and is somewhat lenient by default. It allows a number of constructs that are not specifically allowed by either the ANSI C specification or documented in Apple's Objective-C reference guide. I avoided using any non-standard features in XForms code.

## *Model View Controller Design Pattern*

**Design patterns** are "simple and elegant solutions to specific problems in object oriented software design.[11]" There are dozens of design patterns in existence. The **Model/View/Controller** (**MVC**) design pattern is a proven method of designing software that makes use of a graphical user interface. MVC is characterized by three or more classes that separate the application's data model, user interface, and control structure. This design approach was used in Smalltalk-80[12] and Apple encourages its use since the Cocoa framework provides significant support for applications implementing the MVC design pattern.

The chief argument for using MVC is that it supports code reuse. When the user interface and data model are decoupled from an application's control structures, proponents of MVC argue that both the data model and GUI classes become better candidates for reuse either as is or with little modification.[13]

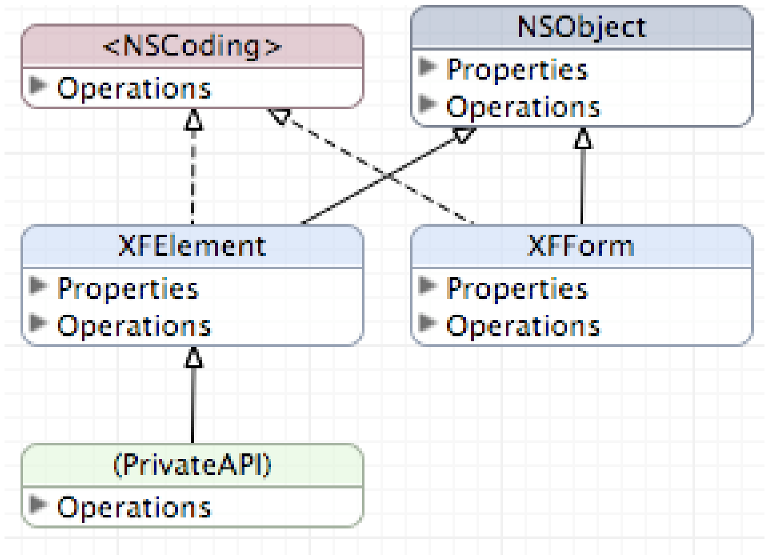
[10] See http://www.gnustep.org. The web site was down when I performed a cite check, but a cached version is available from Google. Installation packages are available for some versions of linux; I attempted installing GNUStep on Gentoo Linux using Gentoo's portage utility, emerge, but I was unsuccessful.

[11] Gamma, et al, p.4.

[12] Ibid referencing and quoting: Glenn E. Krasner and Stephen T. Pope. *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*. Journal of Object Oriented Programming, 1(3):26-49, August/September 1988.

[13] Gamma, *Design Patterns Elements of Reusable Object-Oriented Software.*

**Figure 4: XForms Model Classes**



## Model

Model classes represent application data independent of the format or mechanism by which the application displays that information. The XForms model classes are:

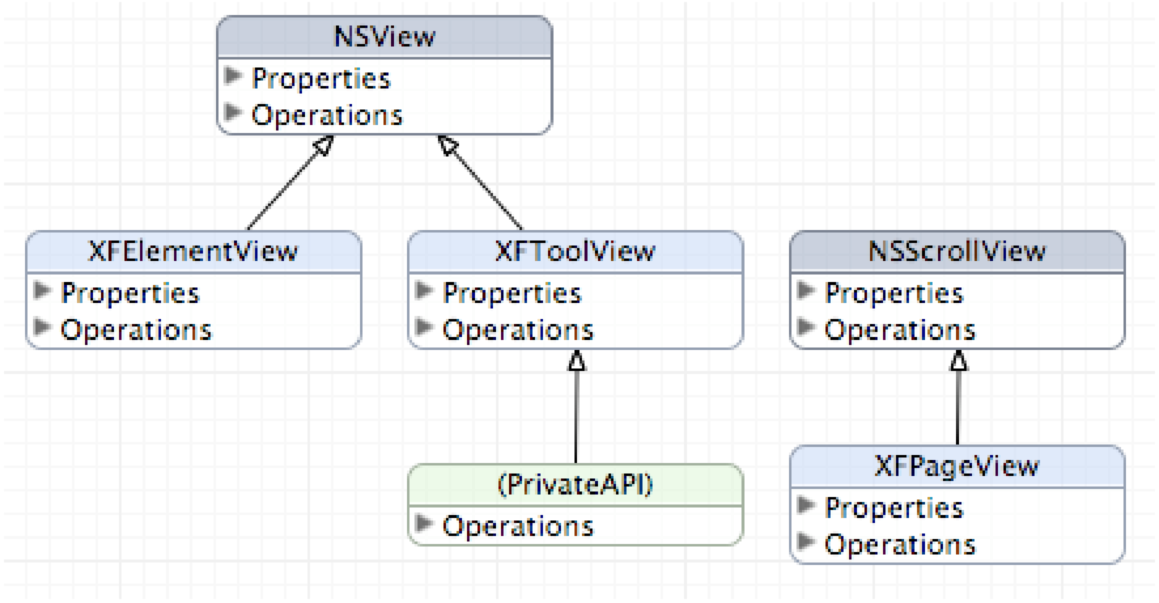
1. `XFElement` – represents a form element and its related metadata.

2. `XFPage` – consists of meta-data related to a page, including its size and the number of `XFElement` objects on the page.

3. `XFForm` – consists of metadata related to the entire form, including its file name, version, number of pages, and a list of each `XFPage` object instance comprising the form.

All `XFForm` model classes inherit from `NSObject`, Cocoa's base class.

**Figure 5: XForms View Classes**



## View

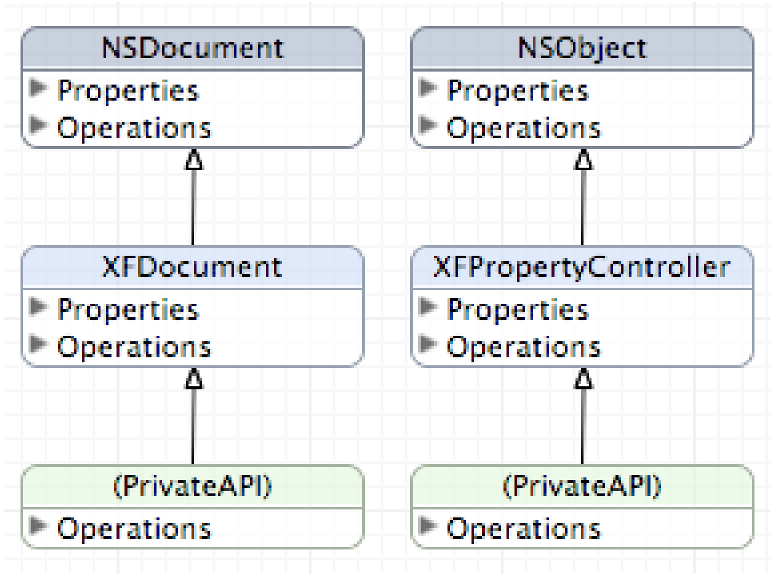
View classes represent the user interface to data. Any data displayed by a view class is an ephemeral snapshot, lasting only as long as the view is available. If data in either the view or the model changes, synchronization methods update the other component if one wants to retain the data update. The XForms view classes are:

1. `XFToolView` – displays a tool pallet of form elements, such as images and buttons, that may be added to a form.

2. `XFElementView` – displays a single element on a page. The display consists of a border and an element from the tool pallet. The border helps the designer align elements and is not part of XForm output. `XFElementView` inherits from Cocoa's `NSView` class.

3. XFPageView – displays a form page. Each instance holds a collection of `XFElementView` objects. This class is a subclass of Cocoa's `NSScrollView` class. `NSScrollView` provides facilities for managing a view that exceeds the bounds of a window.

**Figure 6: XForms Controller Classes**



## Controller

Controller classes coordinate activities between view and model classes. In XForms, the controller classes are:

1. `XFDocument`– maintains a reference to all view and model class instances. It creates each `XFElement` and `XFElementView` instance. In addition, `XFDocument` provides functionality related to printing, loading, saving, and web-form generation. `XFDocument` is a subclass of `NSDocument`.

   As part of its mission to coordinate the model and view classes, `XFDocument` acts as a **delegate** for the window in focus and for the application's menu. A Cocoa delegate is a helper object that is used to extend the functionality of a framework class. When a delegate for a Cocoa class instance is set, the delegate object agrees to implement a mandatory set of methods for the primary object with which it registered and has the option of implementing others as needed. These methods may provide information to the primary object or may act as a notification to the delegate object of an event handled in the primary object. For example, after a window loads and is displayed in Cocoa, the message `windowDidLoadFromNib:` is sent to the window's delegate, assuming that an object registered itself with the window.

   A delegate receives a selected set of the messages sent to the primary object and a handle to any objects needed by the delegate to alter the parent object's behavior. When a user selects a menu item or performs selected actions within the key window, `XFDocument` receives a message. In response, `XFDocument` often dispatches a message to one or more model and view classes.

2. `XFPropertyController`– handles views, displaying `XFFormView` and `XFElementView` properties that can be altered by a user. These properties include page size, element name, size, and location.

## MVC Class Interactions

`XFDocument` maintains a handle to the primary model (`XFForm`) and view (`XFPageView`) class instances. `XFDocument` may therefore dispatch messages directly to either instance by calling a public method in the destination class. However, the model and view classes do not have a handle to each other and cannot directly communicate in the same manner as `XFDocument`.

Cocoa provides an intra-application messaging system implemented primarily through the `NSNotification` and `NSNotificationCenter` classes. Each Cocoa application has a default notification center that allows any object to send a message, known as a notification, to a default notification queue. A notification consists of a name, a handle to the object sending the notification, and, optionally, a dictionary object containing additional information. Any object may register with the default notification center to receive messages by specifying the notification name, a reference to the object sending the notification, or a combination of both parameters.

XForms uses this notification mechanism to communicate from model instances to objects on the properties panel. Whenever the underlying data model related to a selected element's property is changed, a message is sent notifying the property panel view of the change. This mechanism allows the model to be completely de-coupled from the view. The model, view, and controller classes do not need a handle to each other. Each message is dispatched to the `NSApplication`'s default message center.

To propagate changes from the views to the model, I chose to give each controller class a handle to the model and related views. The controller instances each pass a handle to themselves to the model and view classes. Any changes to the model and view are dispatched directly using these handles. I made this implementation choice in order to compare the speed and implementation issues with both techniques. In hindsight, I would have used Cocoa's notification system and completely disconnected the model, view, and controller classes by implementing Cocoa's `NSKeyValueBindingCreation` category in XForms' model classes.

Cocoa bindings work by registering one object as a key-value observer of another object's data. Anytime that data is changed, the observer object is sent a message containing a handle to the object whose data changed, and optionally, a container object with the changed data. This implementation would have been inefficient in the case of XForms' properties panel since the element properties view content is created and destroyed every time the selected form element on a page changes. Using bindings here would mean sending `bind` and `unbind` messages to a model class for each form element attribute on the view. Even so, I would have used Cocoa bindings in hindsight for two reasons. First, using bindings completely decouples model and view classes since neither need a handle to the other. With bindings, model and view objects communicate with an intermediary—the application's default message center. Decoupling makes it very easy to reuse code without any changes. Second, there is no real penalty for using bindings. Although total message count goes up and there is some delay due to messages passing through an intermediary, all of this happens in a fraction of a second on my 800MHz iBook G4. In each case after a property change, XForms transitions to a state waiting for user input with nothing to do until the user triggers an event message.

### Form Output Options

At present, XForms provides two output formats: paper and HTML. In addition, the form design can be saved in two different formats. One is a binary representation created using Cocoa's `NSCoding` protocol that serializes data output; the other is XML.

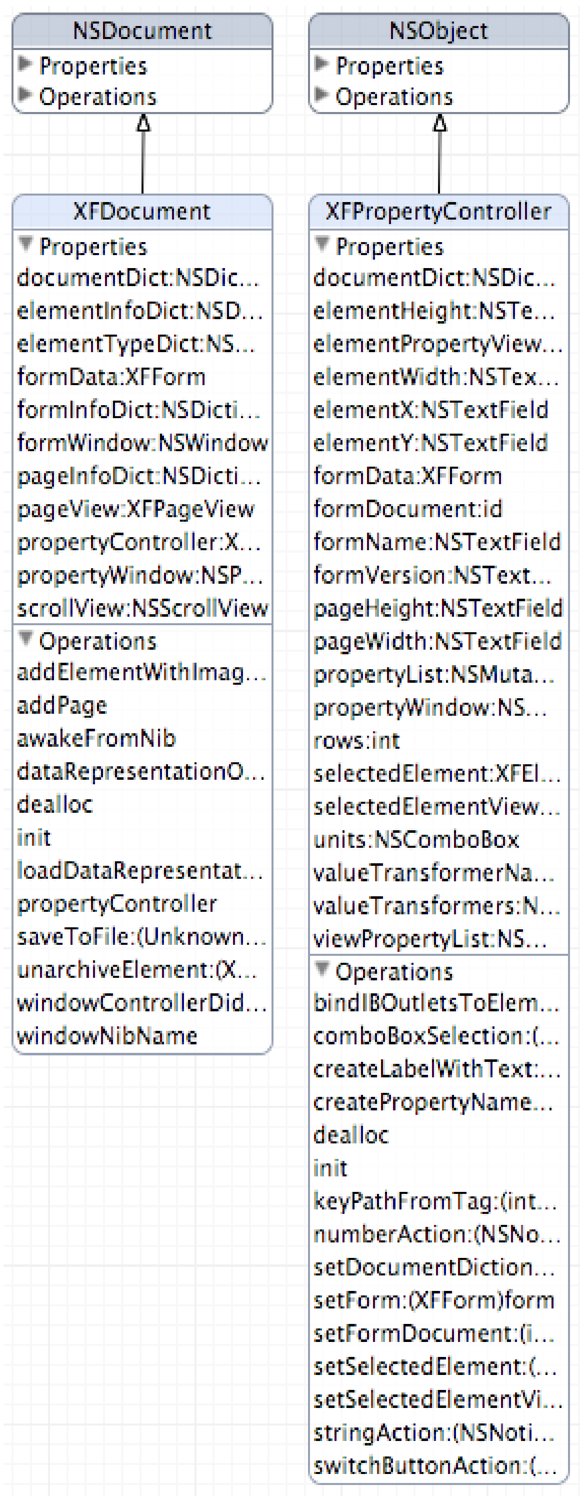
## *Future Extensions and Modifications*

In order to raise XForms to a production st6atus capable of real-world office automation, a great deal of work remains. In particular, the following features are needed:

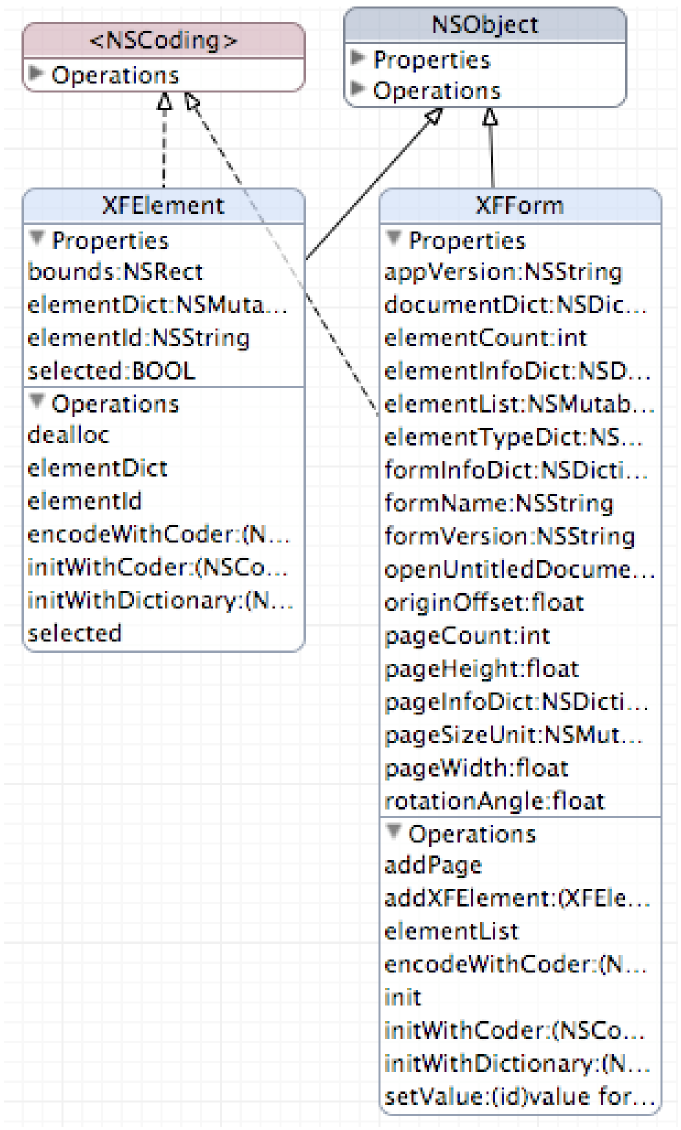
- Database support to capture form output
- Additional design tools (lines, boxes, tables, applets)
- Enhanced form file format support
    - Support for Adobe's "fill-in" format
    - Support for competing products formats
        - JetForms
        - FormFlow
        - Adobe InDesign
- Finer grain control over property values
    - Font selection & properties (color, size, kerning, etc)
    - Element alignment and containment
- Support for more complex, multi-page forms
- Support for user preferences
- User interface enhancement
    - Snap to grid functionality
    - Additional form elements

## Appendix: Detailed Class Diagrams

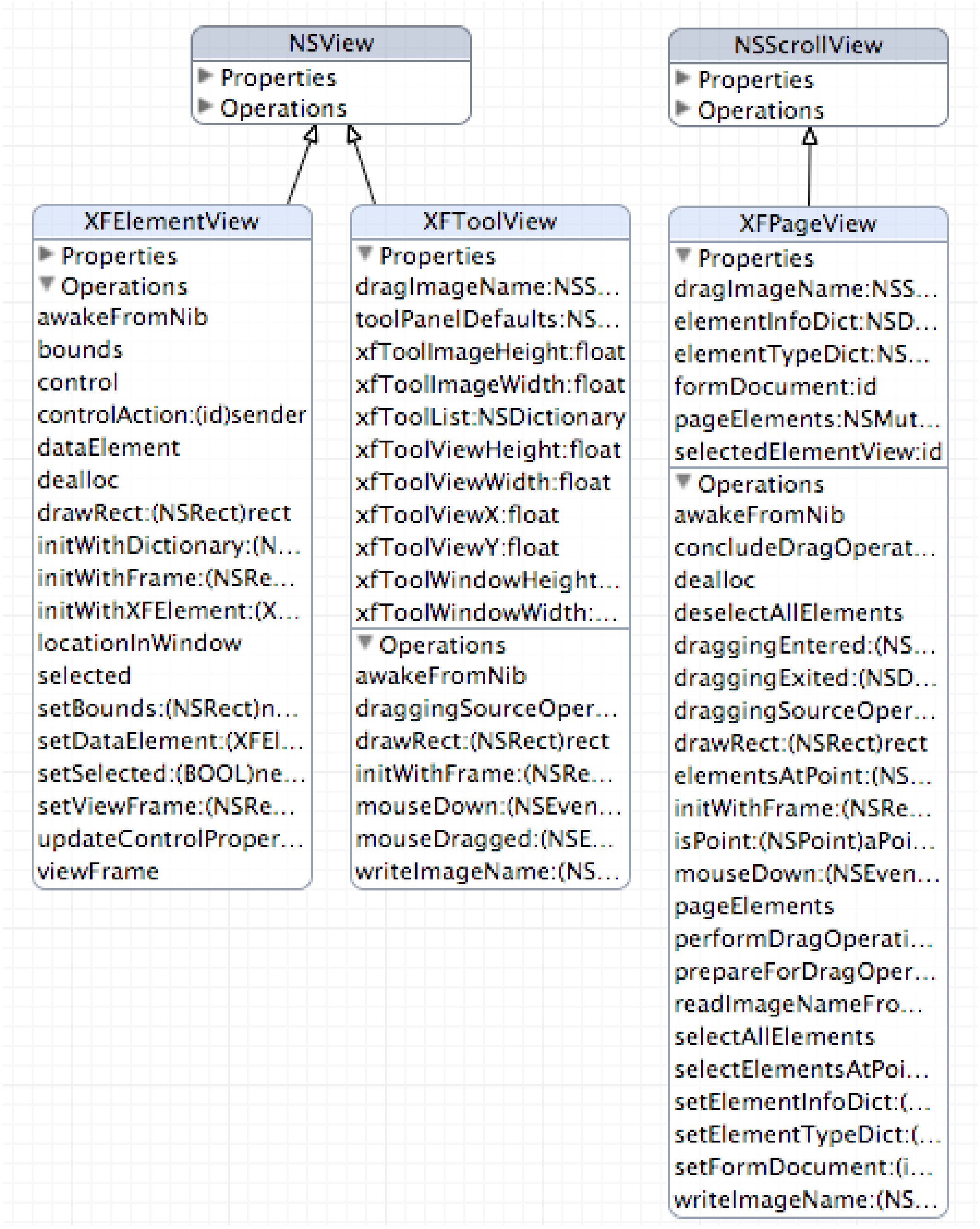
### Controller Classes



### Model Classes



**View Classes**


NSView
Properties
Operations
NSScrollView
Properties
Operations
XFElementView
Properties
Operations
awakeFromNib
bounds
control
controlAction:(id)sender
dataElement
dealloc
drawRect:(NSRect)rect
initWithDictionary:(N…
initWithFrame:(NSRe…
initWithXFElement:(X…
locationInWindow
selected
setBounds:(NSRect)n…
setDataElement:(XFEl…
setSelected:(BOOL)ne…
setViewFrame:(NSRe…
updateControlProper…
viewFrame
XFToolView
Properties
dragImageName:NSS…
toolPanelDefaults:NS…
xfToolImageHeight:float
xfToolImageWidth:float
xfToolList:NSDictionary
xfToolViewHeight:float
xfToolViewWidth:float
xfToolViewX:float
xfToolViewY:float
xfToolWindowHeight…
xfToolWindowWidth:…
Operations
awakeFromNib
draggingSourceOper…
drawRect:(NSRect)rect
initWithFrame:(NSRe…
mouseDown:(NSEven…
mouseDragged:(NSE…
writeImageName:(NS…
XFPageView
Properties
dragImageName:NSS…
elementInfoDict:NSD…
elementTypeDict:NS…
formDocument:id
pageElements:NSMut…
selectedElementView:id
Operations
awakeFromNib
concludeDragOperat…
dealloc
deselectAllElements
draggingEntered:(NS…
draggingExited:(NSD…
draggingSourceOper…
drawRect:(NSRect)rect
elementsAtPoint:(NS…
initWithFrame:(NSRe…
isPoint:(NSPoint)aPoi…
mouseDown:(NSEven…
pageElements
performDragOperati…
prepareForDragOper…
readImageNameFro…
selectAllElements
selectElementsAtPoi…
setElementInfoDict:(…
setElementTypeDict:(…
setFormDocument:(i…
writeImageName:(NS…