

Avoka SmartForm Composer Data Dictionary – White Paper

Why use a Data Dictionary?

Many organizations are recognizing that it is necessary to take a more systematic and rigorous approach to building forms. A form can often be the first impression that a customer has of your organization, and so it's important to design forms that conform to corporate styles guide and standards, and provide the best possible branded user experience. Composer provides this level of rigor through its ability to define templates, style-sheets, and re-usable blocks.

However, it's also very important to apply rigor and structure to the way in which the data entered into the form is collected and represented. In particular it is important to ensure the consistency of data across forms. In order to achieve this, you need some way to define and organize the data being collected, across your entire organization, and use this to drive and inform the design of your forms. This can be achieved using a Data Dictionary.

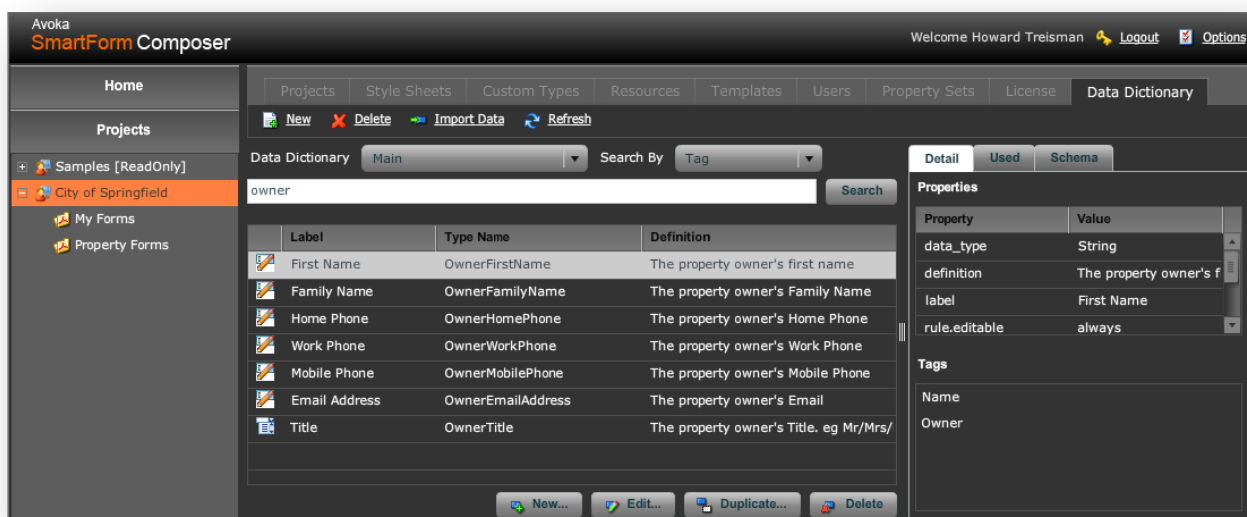


Fig 1 – A Composer Data Dictionary defined for an organization

There are a number of benefits of using a Data Dictionary. These include:

- It promotes consistency between forms – an element defined in the Data Dictionary which appears on a number of forms will always have the same label, tooltip, list of drop-down values, etc. For example,

CustomerMaritalStatus will always be represented by a drop-down list, and will always have the values “Single, Married, Divorced, Separated, Widowed” on all forms.

- It allows you to define the data being collected by your organization in business terms, rather than technical terms. In other words, it defines a common vocabulary. This promotes understanding between different departments or product owners, as well as providing a bridge between the business users and the form developers.
- It defines the way that form data is structured, so that developers have a clear and unambiguous definition of the data that they can use for development of backend systems.
- It optionally defines how the data from the form is stored in back-end systems – often the rules about how a particular piece of data should be collected on a form derives from the way that data is stored in a backend system. For example, FirstName should be no longer than 18 characters.
- It optionally defines how a particular piece of data can be pre –populated into a form from a backend system, to avoid the user having to type in some information that we already know about them.
- It ensures that the same piece of data will always appear in the same XML location in each form – this in turn means that information that has been entered in one form can be automatically used to pre-populate the same information in secondary forms (we call this “cross-fill”), which makes for less typing, fewer mistakes, and happier customers.
- It provides a way for a business or data analyst to have a conversation with the form owner, and communicate about the data being captured, rather than getting bogged down in the look or structure of the individual fields.
- If implemented well, it provides a way of very quickly designing a form based on the data elements that have been identified during discussions with the form owner. This means that the Data Dictionary and the Form Design tool need to be very tightly coupled – you can’t use two separate tools.
- It provides a mechanism for data mining across forms. For example, if I have a field on several different forms bound to a Data Dictionary Element called “CustomerMaritalStatus”, then it doesn’t matter where that field is physically represented on the form, or what it’s called, or how it’s implemented – I can easily find out all forms filled out by any customer whose marital status is “Single” (assuming that’s the information I need).

What sort of information is stored in a Data Dictionary?

The Data Dictionary is NOT meant to store the look and feel of fields that you use in your forms. This capability is already a core part of Composer. For example:

- You would normally define the following field-level properties in the your corporate style-sheet: whether the caption is top or left, the caption reserve width, the fonts and colors, the look of the headings.
- For specific types of fields, you would generally create a custom widget. For example, our Email Text Field has built-in email validation rules. Or you might have an employee number field with a particular display pattern and validation.
- For multi-field types, you would generally create a custom block. For example, an Address block, or a Personal Details block consisting of title, names, telephone numbers and employee id.

The Data Dictionary is more about the different types of data that you collect in your forms. For example, in your organization, you may collect several different email addresses, such as:

- Applicant Email Address



- Alternate Email Address
- Emergency Contact Email Address

Some or all of these email fields may appear on different forms that your organization publishes.

All of these would be mapped to the “Email Text Field” defined in Composer, so they will all automatically use the appropriate look and feel defined by that Field, as well as the validation rules for an email address.

The important consideration is that Applicant Email, Alternate Email and Emergency Contact Email are all fundamentally different “bits of data”, and they have different properties. For example, the Applicant Email Address is mandatory, while the Alternate Email Address is optional. And their tooltips may be different, “Please enter your usual contact email address”, versus “If you provide an alternate Email Address, we will address all emails to both your primary contact address and also your alternate email address.” It’s important that these properties are applied consistently, no matter which form any of these email addresses appear on. Similarly, for your organization, the label used for names should always be “First Name” and “Last Name”, not “Surname” and “Given Name”. The Data Dictionary allows you to define these data collection elements, and ensure that they are applied consistently across all forms.

Another important consideration is the storage and pre-population of data in the forms. For example, you may be storing data from your forms in your CRM (Customer Relationship Management) system. You would store these different email addresses in different fields in that system. Your customers would be very happy if when asking them to fill in a form, you would pre-populate that form from the CRM information. Similarly, once the user submits a form, it would be helpful to know which fields are which, so we can update the appropriate fields in the CRM system. But how do we know which email address from which field in the CRM system should be used for which email field on the form?

The answer is to use a Data Dictionary. The Data Dictionary allows you to:

- Define each unique bit of data that is collected anywhere in your organization.
- Allow you to specify properties about this piece of data, such as its label, tooltip, data-type, default field type, list of allowed values, editable or not, and any other bits of information that you care to record.
- It allows you to map these properties to the actual fields in the form, so that when you drop a field from the Data Dictionary onto a form, the field is automatically populated correctly.
- It allows you to record the source of a field for pre-populating it (from a CRM system or database table or whatever), and where to store the data from this field when the form is submitted.

Principles of the Composer Data Dictionary

There are a number of core principles of the Composer Data Dictionary. Some of these may not necessarily appear intuitive when you first use the Data Dictionary, so we would like to explain these here.

Emergent rather than hierarchical organization

In the past, attempts at classification have often been based on a hierarchical organization structure, also called an ontology or taxonomy. Examples include the Classification of Living Things, and the Dewey Decimal System. However, the problem with this approach is that it requires constant maintenance as new information appears.



The Dewey Decimal System is current in its 16th revision, and the classification of living things is in constant flux as scientists discover new information and genomes are mapped.

In the forms world, any attempt to classify all the information that an organization ever needs to capture is almost invariably futile, because this information is constantly added to and changing, rendering the previous categorization invalid. In general, attempts to solve this problem using a hierarchical structure result in very general purpose approaches, which therefore fail to solve the basic problem they are designed to address.

In the Composer Data Dictionary, we take a different approach. Rather than trying to create a hierarchy, we simply create individual simple elements. The name of the element uniquely identifies it, and remains unchanged forever. We then add descriptive tags to each element in order to classify it. The structure of the underlying elements emerges from the tagging process, resulting in what is known in Web 2.0 terminology as a Folksonomy. (<http://en.wikipedia.org/wiki/Folksonomy>). Over time, Composer will add more sophisticated tag management features such as tag clouds, tag renaming, and tag merging.

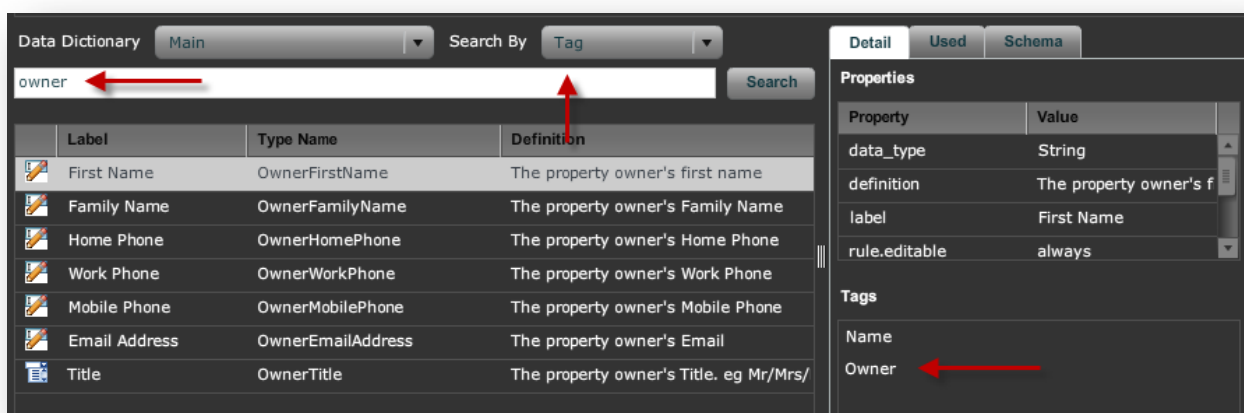


Fig 2 – Searching by tag

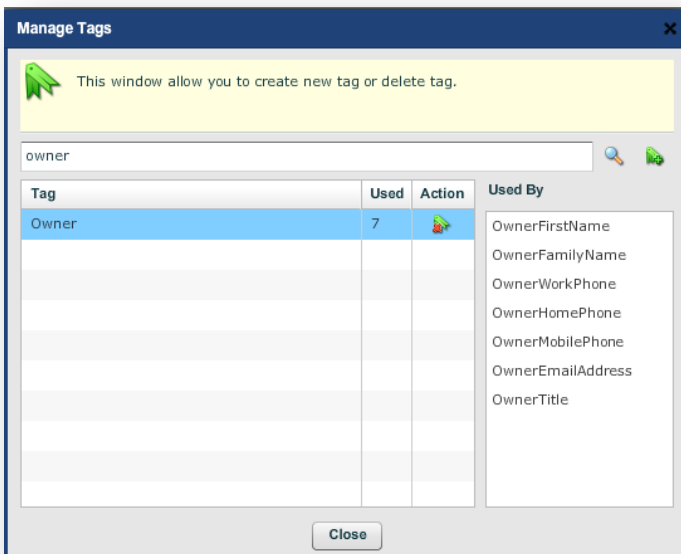


Fig 3 – Tag management

The Composer Data Dictionary does allow the elements to be arranged hierarchically – in fact, this is very important, because ultimately forms store their data in an XML (hierarchical) structure. However, the important thing is that if the XML structure needs to change for any reason, and a particular piece of data ends up being stored in a different location within the XML, this does not change the basic nature of that piece of data – it’s still referenced by the same Data Dictionary Unique name. This is extremely important, because it means that data can be collected over a long period of time, and accommodate changes to the underlying XML data structure, yet still be recognized as fundamentally the same piece of data.

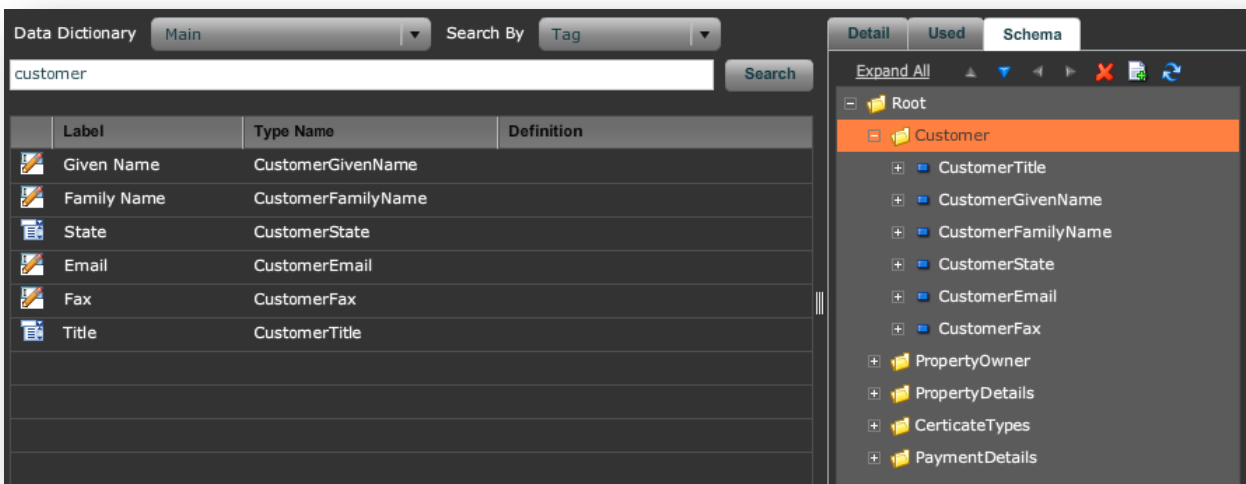


Fig 4 – Mapping Data Dictionary Elements to an XML Schema

Copying Properties to Forms rather than Referencing

When you drag a Data Dictionary element onto a form, it will result in a widget of some type being created, and having several properties applied to it from the Data Dictionary definition. For example, if I drag a Title element to a form, that may be represented by a drop-down list, which has various properties set from the Data Dictionary – its allowed values would be set to Mr/Mrs/Ms/Dr/Other, its label would be “Title”, and its help text would be set to “Please select your usual title.”

One way to implement this would be to have the dropdown list reference (or point to) the Data Dictionary element – each time the form gets generated, the latest property values would be obtained from the Data Dictionary and applied to the drop-down list.

While this approach sounds attractive, in reality it has a number of drawbacks:

- If someone, without your knowledge, changes any of the Data Dictionary elements that you use on your form – the next time you generate your form (perhaps to change a completely unrelated part of your form), those fields would be automatically updated. This effectively means that your form has changed without your explicit knowledge or consent. In larger and more mature organizations, any changes to a form need to be carefully controlled, managed and approved, and automated changes like this would be extremely problematic.
- It also makes it difficult to change something about a particular element on your form so that it deviates from the Data Dictionary. While deviating generally isn’t encouraged, it is sometimes necessary. Ideally it should be possible to use a different value for a property to that defined in the Data Dictionary.

For these reasons, in the Composer Data Dictionary we don’t reference properties from the Data Dictionary; we copy them into the field at the time of creation. This means that once your form has been created, it is completely stable, and will never change automatically just because someone else changed a definition in the Data Dictionary. And it also means that you can manually override those values if necessary.

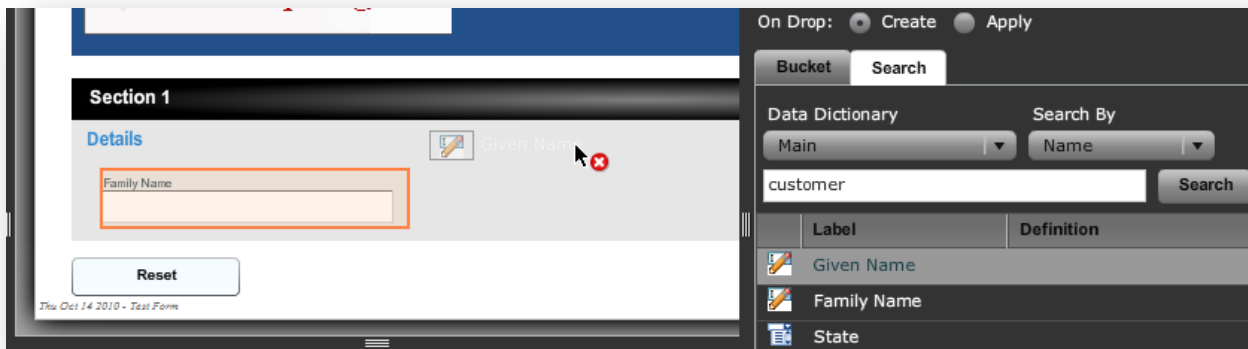


Fig 5 – Dragging a Data Dictionary element (GivenName) into a form results in all the element’s properties, such as caption, help text and others, being copied over into the field.

On the other hand, in many cases, a change to the Data Dictionary definition of an element should be reflected in all forms that use that element. For this reason, Composer also provides a number of tools for verifying and comparing the property values in the forms versus the values in the Data Dictionary. Composer also provides tools

to identify each form that a particular Data Dictionary element appears in, which allows you to explicitly update each of those forms in a controlled and organized fashion.

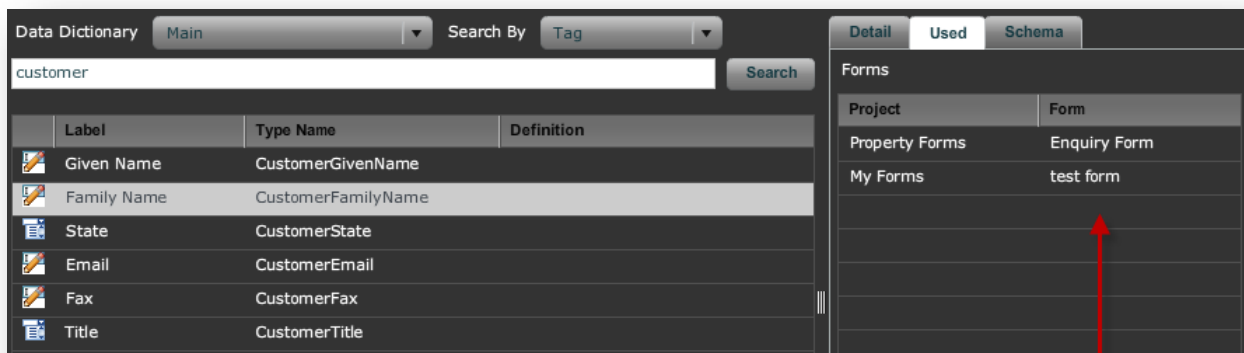


Fig 6 – “Where used” Analysis in Composer Data Dictionary

Duplication rather than a Type System

Most modern programming languages, and some Data Dictionary implementations, use the concept of a type system (also known as a class hierarchy, or abstract base classes). You can define a simple type like “States”, or a more complex type like “Address” (which may include States as one of its child objects), and then re-use those types across multiple instances. For example, ShippingAddress, BillingAddress and ResidentialAddress, all of type Address. One of the advantages of class hierarchies is that if you change something about the base class, this change automatically “ripples down” to all the instances of that type.

This is a very flexible and powerful approach, but it has two disadvantages. Firstly, it is complicated for non-technical people to understand – and the Data Dictionary needs to be very easy for non-technical people, as it is the primary vocabulary for talking about the business. Secondly, it introduces the risk of unwanted and unexpected changes. For example, if a form developer wanted to add an extra field to ShippingAddress, called SpecialDeliveryInstructions, they might be tempted to add this field to the Address base class. However, this would result in SpecialDeliveryInstructions also appearing unwanted in ResidentialAddress and BillingAddress.

In order to resolve this situation, the Composer Data Dictionary does not require you to define types or a class hierarchy. Rather, if you have an existing complex object such as ShippingAddress, you can duplicate it to create an identical object called BillingAddress. These two objects start out being identical, but can be changed independently. There is no type inheritance – each object is just a independent self-contained object. And a change to the ShippingAddress will not result in an automatic change to the BillingAddress. In order to help you to track the various “flavours” of address, we do keep track of which elements were duplicated from other elements, which allows you to compare them.

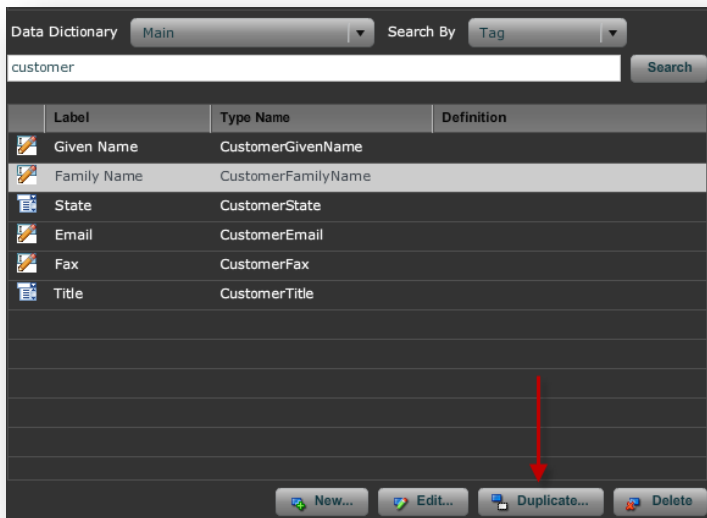


Fig 7 – Duplication

Added-value Widgets

As part of the Data Dictionary design, we have built a number of additional fields and blocks that make common operations easier. For example, a very common pattern is to have a drop-down list with a number of values. One of the values is “Other”, and if you select “Other”, another text-field appears which allows you to enter in the “Other” value. You might have many data elements that follow this same pattern.

We wanted to make sure that this type of pattern would be really easy to implement on a form, without requiring any development of business logic at all. In order to support this, we built this (and other similar Composer objects), and we allow you to map a particular element in the Data Dictionary to either “Regular Drop Down List” or “Drop Down List with Other”. This makes it really easy to create forms with these types of patterns easily and quickly. We also take care of the underlying data binding issues.

Simplicity, Speed and Standardization rather than Flexibility

Two of the primary goals in designing the Data Dictionary are to make it simple and quick to build forms, and to ensure a level of consistency and standardization across forms. This inherently leads to a system that has slightly less flexibility.

For example, if you wanted to create a radio button group with a caption of “Account Type” and three radio buttons called “Credit”, “Savings” and “Check”, you would normally have to create 4 fields – a caption text field, a radio button group, and three radio buttons. For the Data Dictionary, we have created a single composite radio-button group that automatically grows to display as many radio buttons as are needed, as defined by the list of allowed values in your element definition. This means that dragging a single Data Dictionary element to a form automatically does everything in a single step.

However, this does limit your flexibility a little. You are limited to arranging your radio buttons either top-down or left to right – you can’t arrange them in a zig-zag or circular or random pattern. For most forms, this is a small price to pay for the increased simplicity, speed and standardization.

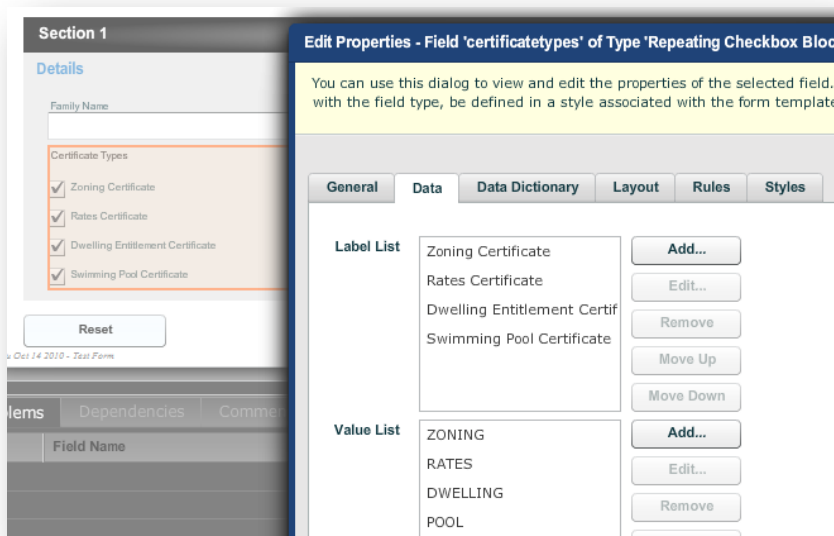


Fig 8 – Certificate Types is a single Data Dictionary element, mapped into a single Composer field. Each of the values is automatically represented as a separate radio button

Another area where we reduce flexibility in favor of simplicity is in the area of XML Schema binding. Particularly for more complex objects, we take care of many of the confusing aspects of data binding for you. This means that the data binding always occurs in a consistent and uniform way, but does limit your ability to bind to arbitrary XML Schemas. In the infrequent cases when you need to support a particular complex schema, you may have to define an XML transform of the data from the form's schema into your schema. In most cases, this is quite straight forward, and well worth the benefit of simplified XML binding in the form.

Composer Data Dictionary Feature Summary

- Separation of roles. A data modeler or data architect creates and manages the dictionary, while a form designer uses the data dictionary to create specific forms.
- Configurable data dictionary element properties. These can be mapped through to properties on Composer fields.
- Tags. Provides powerful tag-based categorization and search
- Vocabulary. Provides a common vocabulary for describing organizational data.
- Complex objects. Provides support for repeating elements and composite elements
- Where-used and Impact analysis. Track a particular element and see which forms uses it. Analyze the impact of changes to a Data Dictionary element.
- XML Schema Mapping. Map Data Dictionary elements to a master XML schema, and automatically have the each form use a sub-set of this master schema.
- Automatic Schema generation. Generate an XML schema directly from the Data Dictionary.
- Supports the way you work. Either build the Data Dictionary first and then create the forms; or create the forms first, and then auto-create Data Dictionary elements based on the fields; or mix and match as needed.