Complex Values in Smalltalk

Thomas J. Schrader
counseling developer
Thomas.J.Schrader@web.de

Christian Haider
Smalltalked Visuals GmbH
Christian.Haider@smalltalked-visuals.com

Abstract Distinguishing between stateful objects and Values has long been recognized as fruitful. Values are universal context free abstractions that prevent side-effects and allow for a functional programming style. Though object-oriented programming languages provide simple Values like Integer and String, more complex Values are usually not supported. We show how to model complex structures as Values in Smalltalk, which allows for a simple and versatile implementation. The resulting Smalltalk systems are simple, clear and easily testable.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects

Keywords Value Objects, serialization, testability, refactoring, functional programming, Smalltalk

1. Introduction

> ≪*Good Design: A system should be built with a minimum set of unchangeable parts; those parts should be as general as possible; and all parts of the system should be held in a uniform framework.*≫ *[Ingalls81, 2nd principle]*

Object-oriented software designers have long realized that conventional object systems can lack information integrity due to their design based on conventional changeable stateful objects. Particularly software workflows with lots of different data options are a common source of failure in conventional object systems and can be hard to design, to maintain and to test.

As early as 1982 Bruce MacLennan[1] called for the recognition of *values* in object-oriented languages as a powerful complementary modeling concept to conventional objects. Properly distinguishing values from objects in order to permit a functional programming style helps to overcome such problems.[2]

MacLennan stated four key properties of values:

- values are abstractions (universals or concepts)
- values are timeless and have no lifecycle
- values have no alterable state and can only be interpreted, but not changed
- values are referentially transparent and prevent side-effects when used on different parts of systems.

Following the evidence given by many practitioners[3] it significantly improves object systems when values are modeled as *Value Objects* and used with value semantics wherever applicable instead of modeling them as mutable stateful objects. There are even comprehensive compilations[4] of Value Objects and other functional modeling patterns in object systems.

However, despite a wealth of research and practice in the last 25 years,[5] the support of Value Objects and value semantics in object-oriented programming languages is still limited to some arbitrary simple cases. The implementation of more specific or more complex Value Objects is still up to each designer and depends on the specific situation, particularly the underlying language.

Here, we present a simple object model and lightweight Smalltalk implementation to build more complex *Value Objects*[6]. We draw upon our observation that any data coming from and going to a system eventually are simple literal values like numbers or strings. Therefore it is always possible to print them.

[1] see [MacLennan82]

[2] In functional programming languages values are the only data option, which turns out to be very beneficial in design situations, where absolute reliability is required when lots of data are distributed around and shared.

[3] see [Bäumer98] for an example

[4] see [Kühne99]

[5] [Riehle06] gives an example of such continuing effort.

[6] In the following we use the capitalized *'Value'* interchangeably with *'Value Object'* to indicate the object model; in contrast we use the lowercase *'value'* for the common meaning of the word.

ACM [to be supplied].

So, a Value is created by a constructor which takes the values of all its instance variables as parameters. Values can print themselves in this constructor format, so that the full definition of the Value is executable and visible at a glance. Since Values are context free and finite, they can always be written as code in a workspace or a (testing) method. Values should naturally never change after creation and do not provide any setters for instance variables.

In section 2 we show some examples and give an overview of the core elements of our implementation. We show a pragmatic way to support Values with a simple generation framework.

Section 3 introduces interface layers, shows how you can explore your domains by modeling them with Values and gives an overview of the applications of Values, which made our Smalltalk systems more reliable and easier to maintain and to test.

In section 4 we address object serialization which is trivial for Values and meta descriptions which we noticed have something in common with our work.

In section 5 we summarize our implementation and shortly discuss pros and cons whereas in section 6 we suggest more enhancements.

— but, let's get to work —

2. Complex Value Objects in Smalltalk

2.1 Values by Example

> «*Examples are an important part of the description of a programming language and environment. Many of the examples used in this book are taken from the classes found in the standard Smalltalk-80 system.*» *[Goldberg83, p. 10]*

Native Smalltalk Values. There are many Values already available in a standard Smalltalk system.

Immediate Objects. Objects like `SmallInteger` and `Character` are represented by the object pointer directly instead of pointing to a location holding the object. We consider them Values because they are handled with perfect value semantics.

So for instance a number like `42` exists as 42 in 4 bytes (including a few bits to specify that the bytes should be interpreted as SmallInteger).

Immediate objects cannot be distinguished (it does not make sense to ask, which *instance* of `42` is used). It is not possible to change `42`, since immediate objects are atomic and do not have internal structure.

When `42` is printed, it will produce a string `'42'`. This string is understood by the compiler to translate it into the SmallInteger `42`. Immediate objects are literal: the compiler will recreate the object from its printed representation so we can use them in source code.

Literal Objects. Other objects are immutable and have a literal representation, which makes them good Values:

- `nil`, `true`, `false`
- Numbers
- `String`, `Symbol`
- (literal) `Array`, `ByteArray`
- `LiteralBindingReference` (only VisualWorks).

`String` and `Array` can also become mutable in which case they are not Values, but ordinary objects.

Instances of real Values can not be distinguished, ideally referring to one single instance. This is true for `nil`, `true`, `false` and `Symbol`s. Whereas Values like `String` or `Array` create equal, but not identical copies.

Value-like Objects. Many more Smalltalk objects are conceptually Values and are often handled with value semantics although they are modeled like conventional objects in the library:

- `Date`, `Time`, `Timestamp`
- `Point`, `Rectangle`, `Association`
- `ColorValue`.

These objects are composed of other objects or Values. Customarily, operations with these objects will return a copy and do not modify the instance. But they have setter methods for their instance variables so that attributes can be changed individually from the outside.

Objects like `Association` may refer to arbitrary objects, not just Values. Only `Association` objects containing Values are Values themselves, otherwise they are conventional objects. The same applies to `Array` and `Dictionary` objects.

Only some of them are literal and can be written in source code directly, like Point (`1 @ 2`) or Association (`#a -> 42`). This is achieved not by special syntax recognized by the compiler, but by using ordinary binary operators to create them. Others can be created sending messages like Rectangle (`1 @ 2 extend: 10 @ 5`).

Often constructors are used to create a Value like:

```
Date
  newDay: day
  monthNumber: monthIndex
  year: year
```

Complex Value Objects. This paper is about complex Values. They are created by constructors, contain only Values and are immutable and literal.

A simple example for a Value is the login data for a user account:

```
User
  name: 'guest'
  password: 'guest'.
```

We can describe a Store connection using another Value:

```
PostgreSQL
  id: #publicCincom
  dbId: #psql_public_cst_2007
  environment: 'store.cincom...'
  user: (User
    name: 'guest'
    password: 'guest').
```

Other Values can reference the Store connection:

```
Package
  name: 'DateField'
  storeId: #publicCincom.
```

A Value can contain lists of other Values:

```
Bug
  description: 'slow...'
  urgency: #high
  changes: (Array
    with: (IssueEntered
      who: #Christian
      when: (Date d: 15 m: 2 y: 2007))
    with: (IssueEstimated
      who: #Christian
      when: (Date d: 15 m: 2 y: 2007)
      hours: 3)
    with: (IssueVersionAssigned
      who: #Christian
      when: (Date d: 15 m: 2 y: 2007)
      version: (Version major: 2 minor: 9))
    with: (IssueVersionAssigned
      who: #Christian
      when: (Date d: 8 m: 8 y: 2007)
      version: (Version major: 3))).
```

Or it can consist of other more or less complex Values:

```
Charttext
  style: (Textstyle
    font: #{SmallCharts.Helvetica}
    size: 12
    color: (CmykColor
      name: 'BN_Blau'
      cyan: 1
      magenta: 0.3
      yellow: 0
      black: 0.3))
  string: 'This is a String'
  position: 5 @ 10
```

All Values presented here can print themselves in such an indented format, so that the structure and all details are visible at a glance.

2.2 The Elements of Values

≪Objects: A computer language should support the concept of "object" and provide a uniform means for referring to the objects in its universe.≫ [Ingalls81, 5th principle]

Values (with a capital 'V') are real objects.

A Value is fully specified by its class and the definition of its content. In general, we define Values as immutable *composites* that can hold other Values in their instance variables. We define a different class for each kind of Value giving it unique behavior.

As we saw in section 2.1 before, we can also directly use some native atomic standard Smalltalk objects as Values. See the next section 2.3, how we convert such Values into a form consistent with our Values model.

We distinguish between three functionally different types of instance variables regular objects can have:

- constant — The content of a constant instance variable is initialized once and cannot change.
- variable — Variable instance variables can freely be assigned from external via setters.
- cache — Cache instance variables initialize lazy on first access. Their content is never modified externally. They can be cleared.

In contrast Values have *attributes*, which are *always* stored in constant instance variables, which can only be initialized with Values. All attributes are initialized at once when the Value is created and can not change afterwards. Values are immutable.

A Value Object satisfies the following criteria:

- attributes contain only Values
- attributes cannot change
- Values should not be distinguished
- Values have a literal representation.[7]

Values of a specific class containing the very same contents should not be distinguished. Therefore, we define equality (`Value>>=`) as the identity of the class and the equality of all attributes.

Values are regular objects — just simpler. By disallowing modifications after creation, we deliberately constrain the full power and flexibility of objects. Systems are complex because of changing state. The state of a regular object depends on its history. By rejecting modifications, Values become trivial objects.

Still, Values are regular objects. We arrange them in arbitrary class hierarchies below a root class `Value` and enrich them with functions and utility methods, just like any object. Also on the class[8] side, any additions like class variables and extensions are possible and common throughout our systems.

Tree-Structure. Complex Values can be seen as top-down trees of Values (composites). Since Values can

[7] A literal representation is not a strict requirement for Values, but this useful property is half the point.

[8] The class of a Value is no Value itself.

only contain Values and can be created only with constructors taking all arguments at once, it is not possible to create recursive or cyclic structures. Values are always concrete, finite and complete.

Attributes in Detail. Attributes can be either a single Value or a collection of Values, which only contain Values as elements. We use `Array` and `Dictionary` for our implementation.

We distinguish four different kinds of attributes:

- *constant* attributes hold a Value of a specified class or `nil` – The attribute is required and must be present in all constructors. The class of the attribute is not checked in the current implementation. The class information is only used to add a comment to the accessor and to generate an example.
- *optional* attributes are constant attributes with a *default* (see below)
- *sequence* attributes specify an `Array` – The elements of the sequence must be Values, but their class is not specified. There are no runtime checks for type compliance as well. Sequence attributes are always optional and default to an empty array `#()`.
- *map* attributes specify a `Dictionary`[9] – Like a sequence, the `Dictionary` must only contain Values (as keys *and* as values), but no classes are specified nor are checks performed. It defaults to an empty `Dictionary new`. We prefer an `OrderedDictionary` to make it better reproducible and to support testing.

Defaults. Attributes can either be a Value or `nil`. Often it is useful to have optional attributes, which provide a default Value when it is not explicitly set. This increases the expressive power and allows for modeling simple domain constraints with mandatory and optional attributes.

The dependencies between default attributes can be complex. For example `Time`: when only hours are given, minutes and seconds can default to 0. But when hours and seconds are specified, minutes have to be specified as well.

Since optional attributes can be omitted when creating a Value, there must be a constructor without that attribute. A Value class with several optional attributes needs constructors for all combinations.

The example

```
<constant: #id class: #{Symbol}>
<optional: #maybe class: #{Symbol}
  default: '#sure'>
<sequence: #list>
```

generates the full constructor

```
#id:maybe:list:
```

and the optional constructors

```
#id:maybe:
#id:list:
#id:
```

Literal Values. Values should have a literal representation from which they an be created so they can be literally written into source code. A Value should be able to print itself in that form. This is similar to the self-evaluation property of objects in Squeak.

Many objects in standard Smalltalk have this property; immediate and immutable objects (Number, Character, String, literal Array, Point etc.) print themselves as source code. This is supported either by the Smalltalk syntax recognized by the compiler or through binary operators or message sends.

The `Value` class defines the method `asSource`, which creates a string with the constructor expression and all arguments. This expression can be evaluated to recreate the Value or used in a method to return the Value. We take extra care to produce nicely formatted source strings with proper indentation to clearly show the structure of the nested Values.

The standard Smalltalk method `Object>>storeOn:` also produces source code strings of arbitrary objects, but uses `instvarAt:put:` to set the attributes instead of printing meaningful constructors.

Limitation: References. Values are simple and context free. They are not as expressive as conventional objects. While those can (and often do) form a web of directed references with recursions and cycles, Values can not. In order to use references, these have to be modeled explicitly. It is often possible to use symbols or ids to refer to another object. This requires additional infrastructure and leads to higher complexity.

2.3 The Creation of Values

≪Give away your best ideas.≫ [Gerald M. Weinberg]

A Value is fully specified by its class and the definition of its attributes. We specify the attributes in the class side method `#localSpecification` using pragmas:

```
<constant: aSymbol class: aClassRef>
<optional: aSymbol class: aClassRef
  default: aString>
<sequence: aSymbol>
<map: aSymbol>
```

For example, the Value class definition:

```
Example class>>localSpecification
  <constant name: #id class: #{Symbol}>
```

[9] Map attributes with the same contents but different order should not result in different Values.

```
<optional: #active class: #{Boolean}
  default: 'true')>
<sequence: #items>
<map: #properties>
```

allows Values like:

```
Example
  id: #Me
  active: false
  items: #(3 #b 'c')
  properties: (Dictionary
    with: #key -> 'Value')
```

The full specification is an ordered list of attribute definitions.

All `#localSpecification` methods of all superclasses in the Value hierarchy are combined to get the attribute definitions in hierarchy order. The ordering is reflected by the order of the instance variables. Attributes can be redefined in subclasses. The class or type may change, but the order of attributes does not.

Code Generation. From the specification of a Value class, we generate all necessary support methods, namely constructors, initializer, accessors, the printer and an example.

We first check whether the attribute classes are Value classes. If so, the following is generated:

- the *class definition* with ordered instance variables
- one *full constructor*, which takes arguments for all attributes
- $2^n - 2$ *optional constructors* [10] for all default attributes n – For each default attributes, we provide an own constructor leaving out that attribute.
- one *initializer*, which sets all instance constants at once when it is called by the constructors – Besides the accessors, this is the only method accessing the instance constants directly. This is safe, because all attributes are set at once. There is no proper way to alter the values of attributes separately.
- one *accessor*[11] for each instance constant – Except for `constant`, all other attribute types provide a default Value. The class of the attribute is inserted as method comment.
- one *print method*, which produces a formatted source string containing a constructor – Default attributes are omitted unless they are set to a Value.
- one class method `#example` containing the source of a generated example[12] using the full constructor – Since all Values have an example, it is always possible to generate an example for a complex Value, but usually we adapt this later on to provide a good general purpose example to be used in unit tests.

Convert Standard Classes to Value Classes. If objects of an existing class only hold Values in its instance variables, the object can be extended to behave like a Value. We therefore provide constructors and a print method, which uses them.

Look at the class `Time` as an example. `Time` objects are usually constructed by `#fromSeconds:` or by sending `#now` to the class. `Time` objects print themselves according to the local setting.

We created an additional constructor:

```
Time class>>h: hours m: minutes s: seconds
  ^(self new hours: hour
              minutes: minute
              seconds: second) beImmutable
```

And the print method:

```
Time>>asSource
  ^self class name asString,
    ' h: ', self hours printString,
    ' m: ', self minutes printString,
    ' s: ', self seconds printString
```

which produces `'(Time h: 22 m: 15 s: 0)'` when sent to a time object holding 10:15 pm. This can be evaluated to reproduce a copy of the original object.

To recognize a class as Value class outside the Value hierarchy, we provide the method `literalExample`

```
Time class>>literalExample
  ^self h: 14 m: 36 s: 29
```

All classes implementing `literalExample` are accepted as Value classes and can be used in attribute specifications.

2.4 Implementation

As programming with Values is basically a programming style with certain conventions, we implemented a certain generation support, which takes care of tedious tasks, like writing constructors and accessors, and we built elaborate printing support into the Values system.

Our implementation is not yet mature. It grew out of laziness to avoid writing all the trivial methods by hand. Although the generator approach is not perfect and editing is sometimes necessary, it has helped making use of Values a lot.

The generator takes special care of redefining Values. In most cases, it will change or remove methods affected

[10] This is unfortunate, but Smalltalk does not support optional arguments.

[11] Accessors are the best place for comments and explanations since you trip over them when debugging

[12] For interactive programming and testing, we always like to have live example objects available.

by a change. Ideally, one need only to work with the *specification* of the Value — the generator takes care of the rest.

We wanted to have as much functionality as possible implemented as a generic framework with a root class `Value` offering all its capabilities to subclasses. While *equality* and other services are implemented generally in class `Value`, some infrastructure has to be generated for each individual Value class: *constructors*, *initializer*, *accessors*, *printer* and *example*.

The design of the Value support was guided by the distinction between runtime and development. During development, Value classes are defined and the support methods generated. In the runtime it is only necessary to create and use Values. Therefore, the constructors, initializer, accessors and printer are packaged in the runtime while the specification and example are only needed in development.

The printer is the only method generated, which could be implemented generically, if the specification would be available at runtime.

Currently, the specification is implemented using pragmas, which have to be assembled from the hierarchy in order to be used for the code generation. This is slow compared to direct object access. This is not important in the current implementation, since the specification is used only once for generation.

The pragma-based implementation is probably inadequate when the specification is used all the time for printing or type checking on creation (see section 6).

Limits of our Implementation. There are few limits imposed by Smalltalk to our implementation.

The number of (Smalltalk) literals per method is limited. This restricts the size of a Value to be stored in a method. VisualWorks recently raised that limit from 256 to 16.777.216.

The length of a method selector is limited. This puts a limit[13] on the number of attributes, which can be specified in one Value class, because the initializer and the constructors easily can reach a greater length.

Interestingly, in VisualWorks[14] this limitation comes from two different sides. First from the compiler, which compiles such a method but crashes the image when the method is called. The other limit is from Store, which simply disconnects with an error when the selector is too long.

3. Using Values in System Design

≪*There really is nothing definitive to say about the "right way" to choose objects.*≫ *[Goldberg83, p. 8]*

[13] The number of possible attributes also depends on the size of their names.

[14] The two limits are different with the Store limit being lower.

The introduction of Values to our systems had the effect that large parts of the software could be trivialized thus reducing complexity.

Values conceptually are not well suited for database applications where objects are often changed, nor for the user interface layer where users directly interact with objects.

Here, we show examples where we could apply Values beneficially: modeling interface layers with Values, testing systems, configuring with Values and using Smalltalk as Value database.

3.1 Interfaces

Values are perfectly suited for interfaces between the system and its environment. By introducing interface layers we can represent any incoming and outgoing data as Values. Data processing would then consist of mapping some Values to some other Values.

Value interfaces are very useful for modular testing. Each side of the interface can be tested separately without the need for the other side to be present.

The entities of a system modeled as Values can be serialized easily, since they do not contain cyclic references. Nor must we be afraid of side effects when we consistently treat data as Values.

Request — Response. External interfaces are either for clients outgoing calls (or requests) and incoming results (or responses) or for server incoming requests and outgoing responses.

No matter, in what form the data are exchanged with the outside, there will never be Smalltalk objects but always simple values like strings or numbers in some structure. Since all external data are values, it is simple to define specialized Value Objects to model a request and a response thereby adapting the raw data to the model to make it usable.

Example for a Client Call. Market data of shares can be retrieved from financial news agencies, like Bloomberg. To request the name and the last price for Google shares from Bloomberg, you call:

```
bloomberg
  dataFor: #('GOOG US Equity')
  fields: #(#NAME #PX_LAST)
```

where `bloomberg` is a provider object doing the communication. The result looks like:

```
#(#('GOOGLE INC-CL A' 414.06)).
```

For the Value interface we define a new Value class:

```
Request class>>localSpecification
  <sequence: #securities>
  <sequence: #fields>
```

so that we could write:

```
Request
  securities: #('GOOG US Equity')
  fields: #(#NAME #PX_LAST)
```

If we carefully chose meaningful names for the Value class and possibly unnamed parameters of the interface, the code can become expressive and understandable.

To connect the Value to the interface we define:

```
Request>>sendTo: aProvider
  ^aProvider
    dataFor: self securities
    fields: self fields
```

To get the result array we can write:

```
aRequest sendTo: bloomberg
```

For the response we define another Value class, which holds the request and the returned data:

```
RequestedData class>>localSpecification
  <constant: #request class: #{Request}>
  <sequence: #data>
```

The request-response Value now looks like this:

```
RequestedData
  request: (Request
    securities: #('GOOG US Equity')
    fields: #(#NAME #PX_LAST))
  data: #(#('GOOGLE INC-CL A' 414.06))
```

To create a `RequestedData`, we add a method to `Request`:

```
Request>>dataFrom: aProvider
  ^RequestedData
    request: self
    data: (self sendTo: aProvider)
```

The `RequestedData` Value has all data from the request-response exchange in a context free form.

Values can be stored and used for various purposes like logging, capture/replay and can be easily mapped or transformed. But most important to us were the enhancements of testability.

Test Interfaces. A great deal of our enthusiasm about Values comes from the fact that our systems became nicely testable once we introduced Values. This was our original motivation for the Values idea.

Value interfaces should be defined just for the sake of system maintenance and testability. They influence several aspects of developing software:

System Testing. To test an external interface, you create a `RequestedData` Value, get hold of a provider object and send the request. The response can then be compared to the expected data.[15] Often, these interfaces are simple and generic, so that testing is of limited value, since bugs would be simple and would occur with the first workspace tests.

Testing the other direction is more useful. Crafted or generic `RequestedData` Values could be defined and fed into the system without the need to be online connected to a provider.

Error Reproduction. It is common practice to print out system stacks at client sites on system failures and log them or send them in as error reports. Unfortunately, stack printouts do not suffice for reproducing data related problems as they cannot reliably communicate data values. It is error prone to reproduce data from error logs by analyzing stack printouts.

To handle data related software problems, we use Values as a representation of online data at client sites. As Values can accurately print themselves, they can be transferred in error reports from client systems to a test system in order to reproduce data workflows in a reliable way.

Data Driven Testing. Directly generating test objects is hard to maintain and impractical for deeply referenced objects. Test data generation should in general be separated from test case execution, but external data sources like spreadsheets either fail to provide complex test objects (and are also bulky to use) or are too far away from the test code to be useful for programmers. Values provide an easy maintainable way to create test objects to be used with generic test cases.

Module Interfaces. Just as external interfaces are beneficially modeled with Values, introducing internal Value interfaces is good for the separation and decoupling of internal subsystems.

The design of Value interfaces can be a challenge, because only context free Values can be exchanged. Often omniscient objects, which provide access to other objects, are passed through the system. It is not possible to transfer such objects through Value interfaces. Rather, the essential basic data has to be identified and modeled as Values.

This is advisable for distributed systems, since sending references between systems is difficult to maintain and often too slow. Values can be used directly with Opentalk Smalltalk-to-Smalltalk connections, since Values always transmit themselves by value, not by reference. Therefore, interfaces modeled as Values can be turned into remote Opentalk interfaces very easily.

Redesign for Values. In order to introduce Values in existing systems to profit from the benefits, some redesign might be necessary.

Consider a server getting requests and creating responses. As soon as a complete request is received, a Value can be created with all that data. The processing and response generation would use that Value. In

[15] This may not be trivial when time based data are involved

the end, a response Value is created with all containing data including the request. The response Value then sends itself over the connection.

This approach differs from other patterns. Often, a response object is created as soon as the request was received. During request processing, the response object is filled with the appropriate data. In the end, the response is complete and sent (with streaming, sending starts before the response is complete).

With Values this is not possible, since they can only be created with all attributes at once. This may lead to a redesign where the construction of a response is broken down to separate computations. This kind of redesign can be healthy as dependencies are cleared up.

Interface Design. All external interfaces are Value-friendly because, in the end, they consist only of structure and primitive data, which can be used for Values. For our interfaces it has been beneficial to model the Values as close to the external representation as possible. For example if a time stamp comes in as `'2009-06-01 12:09:03'`, this string is kept in a Value and no pre-processing or conversion is done. Decoding this data is much better done by the Value itself.

For prototyping it is recommended to create a comprehensive interfacing Value with all external data available, whether you need them now or not. Often, we use the API documentation directly and edit it to define the Value specification. When the interface settles, unnecessary attributes can be removed and the Value class can be regenerated.

3.2 Configurations

Many systems use configurations or specifications to define the setup of a module or subsystem. Configurations may be used by the programmer internally like defining a user interface window with a `#windowSpec` or to setup an Opentalk connection.

Other configurations must be changed by the user or an installation procedure. They often come in the form of INI- or XML-files. Examples are the setup for database connections like Store and other settings in the VisualWorks settings tool, which can be saved as XML.

Values, especially those defined in some class methods are a good representation of configuration settings. A Store connection for example can be defined like:

```
Store class>>publicCincom
  <store>
  ^PostgreSQL
    id: #publicCincom
    dbId: #psql_public_cst_2007
    environment: 'store.cincom...'
    user: (User
        name: 'guest'
        password: 'guest').

Store class>>system
  <store>
  ^Access ...

Store class>>smallCharts
  ...

Store class>>private
  ...
```

These Store Values contain all necessary data to define the Store settings in the VisualWorks settings tool. And, since the Values *live* in methods, the are managed by the regular Smalltalk tools.

All benefits of having Values in code are available:

- directly accessible for testing or in workspaces
- selectors and classes can be referenced
- the structure and details are presented in an easily readable way
- the source code is version managed
- other Smalltalk tools like syntax highlighting and completion are available.

While external configurations are often stored in INI- or XML-files, users should not edit them. We find it more user friendly to offer an editor and use external files only for the transfer to other images.

If the transfer is the only purpose for the external files, it would be easier to have Values in methods. Then the configuration would be part of the source code and could easily be shared between images with the regular mechanisms like Store or parcels.

3.3 Explorative Programming

Early design studies[16] can be done using specific exemplars of core entities of a new domain.

For explorative programming and prototypical engineering, Values are very useful. Like with testing software, exploring a new domain is best done by playing with it. Different modeling approaches should be tried with a prototypical implementation. The model must be lightweight and easy to change. When defining a new model class, it is always useful to have real instances at hand for debugging and testing.

The model instances should not be meaningless examples or dummies, rather they should be real instances as they appear in the normal system. For modeling it is often sufficient to have a handful of instances of each model class to implement and test the logic. If the domain only has few instances, it can be modeled completely with instances defined in code.

[16] [Beck07] recently set a good example. Also, read [Tönne07] for an interesting adaptation of the idea for the rest of the world.

This approach suggests using Values for the model. Values can always be represented by code and so, methods can be defined, which return (an instance of) a Value. These methods are usually unary class methods of the Value class.

That way the Smalltalk environment can serve as a nice lightweight (embedded!) database for Values.

Experiences with a Real Example. Some weeks before the european soccer championship EURO2008, Christian started a project[17] for his friends to bet on the scores. One motivation was to start working with Seaside and Glorp. The modeling was done with Values.

In the tournament games are played between teams in stadiums at certain times. We define

```
Team class>>localSpecification
  <constant: #name class: #{String}>
  <constant: #kürzel class: #{String}>
```

and defined all 16 teams as methods:

```
Team class>>esp
  ^self name: 'Spanien' kürzel: 'ESP'
```

... (14 more)

```
Team class>>sui
  ^self name: 'Schweiz' kürzel: 'SUI'
```

Now we can access for example the Swiss team simply by writing `Team sui`.

The games where fought in eight different stadiums:

```
Stadion class>>localSpecification
  <constant: #name class: #{String}>
  <constant: #stadt class: #{String}>
  <constant: #land class: #{String}>

Stadion class>>basel
  ^self name: 'St. Jakob-Park'
        stadt: 'Basel'
        land: 'Schweiz'
```

... (6 more)

```
Stadion class>>bern
  ^self name: 'Stade de Suisse'
        stadt: 'Bern'
        land: 'Schweiz'
```

Now we can define games:

```
Spiel class>>localSpecification
  <constant: #nummer class: #{Integer}>
  <constant: #runde class: #{Integer}>
  <constant: #team1 class: #{Team}>
  <constant: #team2 class: #{Team}>
  <constant: #anstoss class: #{Timestamp}>
  <constant: #stadion class: #{Stadion}>
  <constant: #ergebnis class: #{Point}>
```

```
Spiel class>>spiel01
  ^self
    nummer: 1
    runde: 1
    team1: Team sui
    team2: Team cze
    anstoss:
      (Timestamp d: 7 m: 6 y: 2008 h: 18)
    stadion: Stadion basel
    ergebnis: 0 @ 1
```

... (22 more group games)

```
Spiel class>>spiel24
  ^self
    nummer: 24
    runde: 3
    team1: Team rus
    team2: Team swe
    anstoss:
      (Timestamp d: 18 m: 6 y: 2008
        h: 20 m: 45)
    stadion: Stadion innsbruck
    ergebnis: nil
```

The finals are interesting, since the teams are not determined in the beginning:

```
Spiel class>>spiel25
  ^self
    nummer: 25
    runde: 4
    team1: Gruppe a sieger
    team2: Gruppe b zweiter
    anstoss:
      (Timestamp d: 19 m: 6 y: 2008
        h: 20 m: 45)
    stadion: Stadion basel
    ergebnis: nil
```

The teams are determined by the ranking in the qualifying group. A group is defined as:

```
Gruppe class>>localSpecification
  <constant: #nummer type: #{Integer}>
  <sequence: #teams>


Gruppe class>>a
  ^self
    nummer: 1
    teams: (Array
      with: Team por
      with: Team sui
      with: Team cze
      with: Team tur)
```

... (3 more)

which allows for the ranking table to be computed.

In the final the teams are defined by the result of the semifinals (`spiel29` and `spiel30`).

[17] We like to model the domain in the user language, so the models in this example are named in German.

```
spiel31
  ^self
    nummer: 31
    runde: 6
    team1: self spiel29 sieger
    team2: self spiel30 sieger
    anstoss:
      (Timestamp d: 29 m: 6 y: 2008
        h: 20 m: 45)
    stadion: Stadion wien
    ergebnis: nil
```

These Value classes implement most of the domain model including all their concrete instances. Two former prototypical Values (Tipper and Tipp) were changed to database objects described by Glorp. One Value (Gruppentabelle) was changed to an object since it needed a cache for the computed result table. The rest (20 classes) was related to the Seaside user interface.

During the tournament, after a game, it was necessary to log into the Seaside server and change the code of the relevant game method to add the result. After a reset to flush the caches, the next request would use the new result.

Another motivation in the project was to try Glorp. This was a mistake for such a small application. While Glorp eases some of the burden of having to deal with a relational database, it is still too cumbersome and requires special attention even with only two tables.

It would have been much simpler to have Tipper (account details for a user) and Tipp (the bet on one game) as Values in class methods. Since these objects are dynamic and created by the user, they have to be automatically recorded and stored. This can be done with a simple method generator, which creates an instance method returning the Value:

```
Tipp>>addAsMethod
  self class class compile: self selector, '
  <tipp>
  ^', self asSource


Tipp>>selector
  ^'tipp',
    self spiel number printString,
    self tipper name
```

resulting in:

```
Tipp class>>tipp21Christian
  <tipp>
  ^self tipperId: 3 spiel: 21 tore: 2
    gegentore: 1
```

The tips can be accessed using the pragma `<tipp>`. The compiled definition is available immediately and is recorded in the changes file. For simple applications, this is a basic, fast and reliable database for Values.

3.4 Other Applications

Values are a good bet for objects which only hold simple elements and which are not supposed to change after creation. Using Values allows several services like logging and replaying to be implemented easily.

Actions and Commands. More complex user interfaces implement user commands as objects, instead of just methods, to allow undoing, recording and scripting (see the Command design pattern). Commands should be modeled as Values: once created and executed, they should never change. Commands should also be context free, since they are used in different situations (redo, replay, etc.). Using Values to implement commands clearly separates creation and execution. The creation of a command Value requires all parameters to be present as concrete, context free Values. The execution context of the command in contrast is not restricted and might involve stateful objects like windows or sessions as executor or execution parameters.

The separation of creation from execution is beneficial for the system design, because it decouples responsibilities. While the creation of commands is usually done in the user interface (which can be seen as a mere editor for commands), the execution is independent and can be triggered by commands from any source (user interface or script).

Examples from Smalltalk are changes and refactorings. Changes record all elementary changes to the system in the `.cha` file for error recovery while refactorings are commands allowing complex changes of classes and methods.[18] Although changes and refactorings are context free command objects as described above, they are not implemented that way. Both implement setters for instance variables (like `className:`). This is not intention revealing, since setters should not be called on existing changes or refactorings. They rather should only be used for instance creation.

Announcements and Events. Introducing announcements to Smalltalk has been a great improvement, since it allows dependent systems to exchange events with real objects instead of just symbols. However, there are no restrictions for the objects carried by announcements. Announcements could be used as Values, i.e. allowing only Values in instance variables, to help decouple system layers from each other (user interface from domain layer or domain from persistence layer). Note that announcements with real objects are also very useful inside layers, like announcements containing widgets in the user interface layer.

Events coming in from the operating system are necessarily Values, since they can only contain simple values, no objects with references, and should not

[18] The concepts overlap and may get unified in the future.

be changed. But, similar to changes and refactorings, events often are implemented as ordinary objects with setters for instance variables which should not change after creation.

4. Related Work

Object Serialization. Object serialization[19] is used to exchange objects between Smalltalk images and even between Smalltalk dialects [20]. The core feature is that references between objects in the serialized object net are preserved. For Values this is not necessary, since Values do not have any references. Therefore, Values implement serialization by using their literal representation (i.e. as source code).

Meta Descriptions. The attributes of Values are specified using a simple description language (pragma declarations). The parameters in the attribute specification are important for the purpose of defining Values.

There are other attribute description frameworks for other purposes. Some appliances[21] can generically specify user interfaces. Very popular is the mapping of objects to database tables where attributes are described with all relevant, sometimes complex properties (like in Glorp).

5. Conclusions

Our approach combines three aspects: Values, literals and defaults. Each aspect has been discussed separately before, but the combination of these features allows for a useful extension of object systems.

We found that a strong distinction between regular, changeable objects and immutable Values can lead to more transparent system designs.

Ideally, fewer conventional objects will do the work and need most of the attention during system design. That parts of a system consisting of trivial Values will hardly need any attention other than initial modeling.

Values are literal, presenting their full structure and all details in one expression. Reading a constructor call in code is sufficient to understand the role of the Value – a deeper look is not necessary, since it is trivial.

Value interfaces isolate modules from each other, since only Values can be exchanged. The transmission of regular stateful objects whose state can change is not possible. Regular objects can only exist inside of modules and cannot leave them. From the outside, only Values can be accessed with the interface of a module.

Our implementation basically provides a code generator and a pretty printer and is simple but useful. Creating and changing Value definitions is almost as easy as it should be. Most repetetive chores have been eliminated making it possible to concentrate solely on the modeling aspects.

Values are good for certain modeling situations, but not for all. Database objects allowing changes to attributes conceptually are conventional objects. Read-only database objects could be modeled as Values if they contain nothing but Values. User interface objects allowing the direct manipulation by the user should be implemented as conventional objects. But module interfaces between the user interface and the domain can be usefully implemented using Values.

The performance of our systems has never been a big issue regarding our use of Values. Many methods are used as method Values, which are often executed. Each time a new Value instance is created and returned. This can be slow in tight loops, but is usually not noticeable. Therefore, we develop without any consideration for performance and do not cache any Values. If the system shows performance problems, it is often easy to add caches to some Value classes — with the drawback of introducing state, which needs to be managed.

6. Outlook

While we used Values very beneficially in our systems for many years, some changes and enhancements are desirable.

Well Known Values. Some Value classes have well known instances. For some colors for example it is awkward to write them as

```
(ColorValue r: 0 g: 0 b: 0)
(ColorValue r: 1 g: 1 b: 1)
```

These special Values should appear literally and print themselves as `ColorValue black` and `ColorValue white`. Other examples are: `Point zero`, `Time noon`.

This could be implemented using a class registry and changes to the printer to use it.

Reflection. We implemented the definition of a Value using pragmas. To create the full specification, all attribute definition pragmas of the class and its superclasses are collected from the class method `#localSpecification`.

Our implementation is slow, but sufficient for generating code. To allow for reflection about the specification at runtime, for example for the printer, it must be implemented differently.

We assemble the full specification in two steps:

1. We get the attribute definition lists from each class in the inheritance chain.

[19] as in VisualWorks BOSS

[20] as SRP or SIXX

[21] see the Magritte framework in [Renggli06] as a useful example for Web development

2. We combine them into one attribute definition list respecting redefinitions.

We currently use pragmas for the first step creating attribute definition Values[22] from them. But we could store the definition Values directly in the `#localSpecification` method. The second step would still be necessary to assemble the full specification.

Changing this could also lift the restrictions imposed by using pragmas. The arguments of pragma messages can only be literal. No message sends are allowed. This is why the defaults are specified as source strings. Using real Values in the `#localSpecification` method would allow the use of Values for the defaults.

It would also be possible to cache the full specification in a class instance variable. This cache must be cleared when a superclass specification has changed.

Type checking. We write the names of the classes specified for `constant` and `optional` attributes as comment in the accessor and use them for example creation. No type checking is performed. Although we normally do very well without static types, in the case of Values that might be disputable.

Implementing type checking is fairly simple, since it needs to be done only on instance creation. Fortunately, there is only one method changing instance variables: the *initializer*. It could be changed to accomodate the checks either generically with reflection or directly generated into the *initializer*.

For `sequence` and `map` attributes, the class of the elements could be specified. Special `ValueArray` and `ValueDictionary` classes could ensure that only Values of specific classes are contained.

Specifying the arrity of `sequence` attributes may be considered, since it is valuable model information.

Object Attributes. In Smalltalk, instance variables are defined with a string containing names. For the definition of Values, we use instance variables for the attributes and enrich them with meta-information like *type* and *class*.

It would be nice if Smalltalk had instance variable definition objects[23] (or Values). The default `AttributeDefinition` would just contain the name. Subclasses could then implement other instance variable semantics and constraints.

Caching Method Values. In systems using Values, there can be many methods, which create and always return the same constant Value. `#example` and typical configuration Values are examples for these methods.

Standard literal objects in Smalltalk code are stored in compiled methods. It should be possible to extend the Smalltalk compiler for Value classes. Constant method Values could be compiled to lazily remember the return value on first invocation in the compiled method.

Pragmas like `<constant>` or `<cache>` could be used to indicate the caching behavior. This is an elegant caching solution for constant Values, because the cache is automatically cleared when the method is recompiled.

A similar solution was proposed by Travis Griggs[24]. He implemented the method `once` to make blocks remember the result of their first evaluation.

If the Value specification was implemented with Values instead of pragmas, the specification method would be such a constant method Value, which could be cached this way. This efficient solution would recommend Values to be used for the specification.

Value Refactorings. Working with Values means modeling. Value classes are defined and attributes are added, changed or moved between Value classes.

While our code generator always takes a full specification and computes all changes, it would be interesting to define higher order refactorings to model even more efficient. One pair of complementary refactorings would be `ExtractValue` and `InlineValue`.

`ExtractValue` could take some attributes from a Value to create a new Value class with them. The attributes of the original Value are replaced with a new attribute for the new Value. Accessors of the old attributes are rerouted to the new Value. `InlineValue` would do the opposite.

Tool Support. Programming with Values is mostly editing Value specifications, the rest is done by the generator. This could be supported by a Value definition dialog and a Value specification editing tool in the refactoring browser.

Inheritance Problem. In regular Smalltalk class hierarchies, subclasses can add but not remove instance variables. With Values, attributes can also change.

This leads to the following problem: If the superclass specifies an attribute as default, constructors will be generated without it. If a subclass re-declares the attribute as constant, the defaulting constructors without the attribute are still available in the subclass.

See for example how the following specification:

```
ValueA>>localSpecification
    <constant: #a class: #{Integer}>
    <optional: #b class: #{String}
      default: 'String new'>
```

results in the two constructors

```
ValueA class>>a: aInteger b: bString
ValueA class>>a: aInteger
```

[22] Attribute definitions were the first Value classes we defined for implementing Values.

[23] as discussed in [Griggs07-2]

[24] see [Griggs07-1]

If we define a subclass `ValueB` of `ValueA` which redefines attribute `#b:`

```
ValueB>>localSpecification
    <constant: #b class: #{Symbol}>
```

we get only one new constructor

```
ValueB class>>a: aInteger b: bSymbol
```

but the constructor `#a:` is still available to `ValueB`.

One could forbid redefinitions of defaulting superclass attributes to constant ones in subclasses (redefining a constant attribute with a defaulting one is no problem) or redefine the impossible constructors with an error like this:

```
ValueB class>>a: aInteger
    ^self error: 'Attribute #b is missing.
    Use #a:b: instead.'
```

The automatic generation of such guarding methods has not been implemented.

References

[Bäumer98] Dirk Bäumer, Dirk Riehle, Wolf Siberski, Carola Lilienthal, Daniel Megert, Karl-Heinz Sylla, Heinz Züllighoven, *Values in Object Systems*.
Ubilab Technical Report 98.10.1, UBS AG, Zürich, 1998, http://www.riehle.org/computer-science/research/1998/ubilab-tr-1998-10-1.html

[Beck07] Hans N. Beck, *Exploratives Modellieren*, in OBJEKTspektrum, online edition 'Requirements Engineering', 2007, http://www.sigs.de/publications/os/2007/RE/beck_OS_RE_07.pdf

[Goldberg83] Adele Goldberg, David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983, Free book at http://stephane.ducasse.free.fr/FreeBooks/BlueBook/

[Griggs07-1] Travis Griggs, *When You Come Back* [on caching values], in 'Objology' Weblog, 2007, http://www.cincomsmalltalk.com/userblogs/travis

[Griggs07-2] Travis Griggs, *Slots all the Way Down* [on object attributes], in 'Objology' Weblog, 2007, http://www.cincomsmalltalk.com/userblogs/travis

[Ingalls81] Daniel H. H. Ingalls, *Design Principles Behind Smalltalk*, BYTE Magazine, August 1981, copy at http://www.cs.virginia.edu/~evans/cs655/readings/smalltalk.html

[Kühne99] Thomas Kühne, *A Functional Pattern System for Object-Oriented Design*, Ph.D. thesis, Darmstadt University of Technology, 1999, http://homepages.ecs.vuw.ac.nz/~tk/fps/fps-sans-escher.pdf

[MacLennan82] Bruce J. MacLennan, *Values and Objects in Programming Laguages*, Computer Science Department, Naval Postgraduate School, Monterey, 1982, ftp://ftp.cs.utk.edu/pub/maclennan/VOPL.pdf

[Renggli06] Lukas Renggli, *Magritte – Meta-Described Web Application Development*. Master thesis, Berne University, 2006, http://scg.unibe.ch/archive/masters/Reng06a.pdf

[Riehle06] Dirk Riehle, *Value Object* [in Java...], in 'More Design Patterns' Weblog, 2006, http://wiki.moredesignpatterns.com/space/Value+Object

[Tönne07] Andreas Tönne, *Exploratory Modeling with SAP NetWeaver*, Cincom Systems, Inc., 2007, http://www.cincomsmalltalk.com/cincomFiles/digest2007/CS070214-1-A4.pdf