How to display an object as a string: ***TypeConverter*** *and* ***PrintConverter***

Bobby Woolf

In part 1 of this article, I described the need for each object to generate a short string that identifies itself. VisualWorks provides two messages to do this: printString and displayString. printString displays the object to a developer, so it specifies the object's class. displayString displays the object to a user, so it should not specify the object's class. In Part 2, I'll talk about two classes for converting an object to a String: TypeConverter and PrintConverter.

TYPECONVERTER

TypeConverter is a class that was introduced in VisualWorks 1.0. It is a kind of ValueModel, a model that contains a single aspect called "value." The converter's subject is itself a ValueModel, so the converter enhances its subject by adding conversion behavior while preserving its value-model behavior.[1] This is an example of the Decorator pattern.[2]

The conversion behavior that a TypeConverter adds is the ability to convert its subject's value from one type to another, and back again. This assumes that the two types are convertible and the conversion is bi-directional.

The primary type that TypeConverters convert to is Text, which is just a fancy string. ParcPlace has already implemented the algorithms numberToText, dateToText, objectToText, and so on.

TypeConverter has two advantages over printString and displayString:

- A TypeConverter does not just display the object as a string; it converts the object to its String (actually Text) equivalent.
- A TypeConverter can convert the String equivalent back into the original type.

TypeConverter was introduced to support InputFieldView. An input field's value might be any type of object. The field needs to display the object as a string, a task that displayString can do. However, if the user types in a new string, the field needs to convert that string back into its value's type. The standard message for converting a string into an object is readFromString:.

A TypeConverter encapsulates displayString and readFromString: together so that a source type can be converted into a String and back again. In the process, it remembers the target's type—whether the new object should be a Number, a Date, etc. It also checks for exceptional conditions, such as the original object being nil or times, when empty-string from the user should be converted into nil. Since TypeConverter encapsulates all of this behavior into a single object, it can easily be reused any time this conversion is needed.

AN EXAMPLE

Here's an example of how a TypeConverter can be used. Let's say you've stored somebody's age, and that it's accessible through a ValueModel called ageHolder. Age is a Number, but you need to display it in a field. If the user types in a new age, you need to convert it back into a Number. This code shows the two value-models you'll need:

```
| ageHolder ageAsStringHolder |
ageHolder:=10as Value.
ageAsStringHolder :=
    TypeConverter onNumberValue: ageHolder.
Transcript cr; show: 'age''s type is ',
    ageHolder value class displayString.
Transcript cr; show: 'ageAsString''s type is ',
    ageAsStringHolder value class displayString.
```

The transcript shows that ageHolder contains a SmallInteger, and ageAsStringHolder contains a Text. The input field's model would be ageAsStringHolder. Any code needing to access the age in its unconverted form would go through ageHolder.

PRINTCONVERTER

The problem with TypeConverter, displayString, and printString is that they all assume that there's only one way to show a particular object as a string. This "one-size-fits-all" approach is often insufficient. A Date can be printed

many different ways: December 25, 1990; 25-DEC-90; Christmas Day. A Time has several choices: 4:00 P.M.; 16:00:00; etc. A Number can be printed with leading and/or trailing zeros: 1; 1.00; and so on. TypeConverter can't handle these formatting choices. As long as it converts to the right type, it's done. Subtleties about what exactly the resulting type should look like have to be handled somewhere else.

PrintConverter is essentially a TypeConverter that has been optimized to display objects to the user as strings. Whereas, a TypeConverter can convert from any type to any other type and back again, and PrintConverter only converts to strings. Like displayString, PrintConverter doesn't even convert the object to its string equivalent; it just displays the object as a string. Like a TypeConverter, a PrintConverter can convert an input string back into the original object's type.

*"A **TypeConverter** encapsulates **displayString** and **readFromString**: together."*

The major advantage PrintConverter has over both TypeConverter and displayString is that it can format the string it displays. You do this by specifying the type of source object to be converted, but also by specifying the format of the resulting string. For example,

```
PrintConverter for: #date
```

will create a PrintConverter that will display a Date using the default format. On the other hand,

```
PrintConverter
    for: #date
    withFormatString: 'd-mmm-yy'
```

will create a PrintConverter that will display a Date using the format specified.

The other advantage of the way PrintConverter works is that a single converter can be used to display a number of objects of the same type with the same format. To display a list of twenty Dates with the format 'd-mmm-yy,' you only need one PrintConverter for the whole list. As the list prints each Date, it runs the Date through the PrintConverter, which returns the formatted string for that Date. To perform a similar conversion using TypeConverters, you would need twenty TypeConverters, one for each Date.

Ironically, the only widgets that use PrintConverters are input fields, combo boxes (which, of course, contain input fields), and those data sets that contain input fields and/or combo boxes. SequenceView (the List widget) doesn't use PrintConverter. So if you develop a PrintConverter that formats Dates in a special way that you like, you can use that format to display a list of Dates, in a DataSetView but not a SequenceView. To use that format in a List widget, you have to implement a method, such as Date>>displayStringSpecialWay, and set the SequenceView's displayStringSelector to displayStringSpecialWay.[3] So now you have the same format implemented in a special instance of PrintConverter for Dates and a special method in Date. I would prefer to only implement this code in one place, not two.

WHERE PRINTCONVERTER IS USED

You're already using PrintConverters, even if you don't realize it. In the Painter, when you specify the properties for an Input Field, two of the properties on the Basics page are Type and Format. What you're specifying is the source object's type (String, Symbol, Text, Number, etc.) and its format ((@@@) @@@-@@@@, 0.00, etc.). This is all of the information needed to set up a PrintConverter. When you open the window, as the Builder creates the Input Field, it also creates a PrintConverter with the properties you have specified. Combo Box and Data Set have similar properties that specify the PrintConverter to use.

As you create your own objects that need to be displayed as strings, I suggest you create new PrintConverters to display them. Let's say you have a Money class. You want to be able to display a Money object in an Input Field and get a new one from the user by having him type it within the field.

1. You would need to implement PrintConverter>>initForMoney. Use the corresponding methods for Date, Number, and String as examples of how to implement your methods.

2. Modify PrintConverter class>>for: to add #money onto that big, long case statement.

To make your new Money PrintConverter accessible from the Properties Tool:

3. Modify InputFieldSpec class>>typeMenu to add 'Money' -> #money.

Now the Properties Tool will allow you to specify the type of a value for an Input Field as Money. This will also be available for Combo Box and Data S et widgets.

Modifying the list of formats for one of ParcPlace's types is also simple. See the methods in InputFieldSpec class>formats. For example, to add another string format, modify the method InputFieldSpec class>>defaultStringFormats.

Unfortunately, specifying special formats for your new Money PrintConverter is not very easy. You would need to implement MoneyPrintPolicy as a subclass of PrintPolicy. That ultimately involves implementing MoneyPrintPolicy class>>nextTokenOn: and MoneyPrintPolicy>>print:on:policy:, a task which is not for the faint of heart.

READFROMSTRING:

Earlier I mentioned that TypeConverter (and PrintConverter) use readFromString:. If you don't know what this method does, you'll need to learn so you can implement your own converters.

Object class>>readFromString: is sort of the opposite of Object>>printString. The implementor in Object really only works if the string contains a literal or a store-string (see Object>>storeString).[4] However, implementors in more

specialized classes work well because they can assume that the string represents an instance of that class. For example, Object doesn't know what to make of "April 5, 1982," nor do most classes, but Date is able to make it into a Date.

Just like printString uses printOn: to do most of the work, readFromString: lets readFrom: do everything. Thus you'll never subimplement readFromString:, but you should implement readFrom: in your own classes. Ideally, readFromString: should reverse the process of printString and displayString and should also recognize any formatting that a PrintConverter might throw in. For example, if the printString for a particular Person is "John Smith," you will need to implement Person class>>readFrom: to interpret that string as a Person with that name.

GLOBALIZATION

Globalization (internationalization), a feature added in VisualWorks 2.5, adds a whole new twist to displaying an object as a string. In order to know what string to display and how to interpret a new string, you have to know what language the user speaks and what formatting conventions he uses. This has not been an issue prior to this upgrade, because we always assumed (often inaccurately) that the user speaks American English.

First, ignore globalization when implementing printString (printOn:), storeString and readFromString:. These methods are for developers. Since Smalltalk is written in American English, the methods can assume that they should use that language. In addition I think these methods need to be simple and highly reliable; globalization is an unnecessary complication.

Second, ignore globalization when implementing displayString as well. Just as printString should be simple, so should displayString. Use displayString as a quick-and-simple way to display an object to the user. When this becomes too complicated, such as when globalization is necessary, use a PrintConverter instead.

Third, TypeConverters don't need globalization either. A TypeConverter is only responsible for converting an object's type to or from a string. As long as the object is a String, any effort to format it or translate it into another language is unnecessary. These are responsibilities better fulfilled by a PrintConverter.

Finally, a PrintConverter should use globalization when performing its conversion. Look at the ones that ParcPlace has already implemented for Date, Time, and Timestamp. Their toPrint and toFormat blocks use "Locale current ..." and TimestampPrintPolicy to display a string appropriate for the current location. Similarly, the toRead blocks use "Local, current ..." and TimestampPrintPolicy to read a string from the user. As I mentioned earlier, implementing your own subclass of PrintPolicy is difficult.

A PrintConverter that uses globalization does not use readFromString: to convert a string back into an object. readFromString: does not use globalization, so it is not appropriate for this purpose. Instead, a globalized PrinterConverter uses a LocaleSensitiveDataReader to convert a location-specific string back into an object. Just as PrintPolicy displays an object in a location-specific way, LocaleSensitiveDataReader does the opposite. If your own subclass of PrintPolicy displays an object in a location-specific way, you'll need to implement a subclass of LocaleSensitiveDataReader to read it back.

CONCLUSIONS

Here are the main points in this article:

- Unlike printString and displayString, a TypeConverter can convert a string back into an object again.
- A TypeConverter is useful for converting an object to a String, but doesn't display it as a string very well.
- PrintConverter is designed to display an object to the user as a string. It makes displayString into a first class object.
- PrintConverter is able to format the display string the way the user prefers.
- A PrintConverter uses readFromString: to convert a string back into an object.
- All classes should implement readFromString: (via readFrom:) to convert their print-strings and display-strings back into an instance.
- A PrintConverter should use the globalization framework to format the string for the current location.

*"**PrintConverter** is essentially a **TypeConverter** that has been optimized to display objects to the user as strings."*

I hope you now see that displaying an object as a string is often not a trivial matter. VisualWorks provides several protocols and frameworks to help you. If you learn how to use them well, I think you'll find your system a lot easier to use. S

References

1. Coplien, James O. and Schmidt, Douglas C., Editors. PATTERN LANGUAGES of PROGRAM DESIGN. Addison-Wesley, 1995. "Understanding and Using the ValueModel Framework in VisualWorks Smalltalk" by Bobby Woolf.
2. Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John. DESIGN PATTERNS: ELEMENTS of REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley 1995.
3. Kohl, William and Howard, Tim "VisualWorks List Components" *The Smalltalk Report*, June 1994.
4. LaLonde, Wilf R. and Pugh, John R. INSIDE SMALLTALK, Vol. 1., Prentice-Hall, 1990, Section 6.2.9, "Read/Write Operations: PrintStrings and StoreStrings."

Bobby Woolf is a senior member of technical staff at Knowledge Systems Corp. in Cary, North Carolina. He mentors Smalltalk developers in the use of VisualWorks, ENVY, and Design Patterns. Comments are welcome at woolf@acm.org. or at htpp://www.ksccary.com.