Matisse
The Object Developer's Database

Meta-Classing & Super-Classing in Matisse

A Technical Brief

Matisse Software Inc.

Introduction

This technical brief explores a concept called Meta-Classing and shows how to use this construct to create adaptive code that can run on many different systems. To clarify, the term adaptive code refers to code that will run unchanged, and adapt to different data models.

Change Is the Only Constant

While this is an age-old cliché, every application programmer lives it, especially when creating a generic product that can be customized for each user's needs. Indeed, the popularity of object-oriented programming is owed to its ability to use polymorphism to write adaptive code that applies to many different object types, thereby reducing the cost of maintenance in an ever-changing environment.

For example, consider the case of an application vendor for a CRM system. Ideally, this vendor would like to have a generic application that could be customized to an industry type, e.g., airlines, hotels, or retail. Each of these industries may have a distributor for the software who would further customize it for particular sub-industries. Why? Because the primary CRM concerns for a tropical resort are very different from those catering to business travelers. Figure 1 shows a representation of this chain.

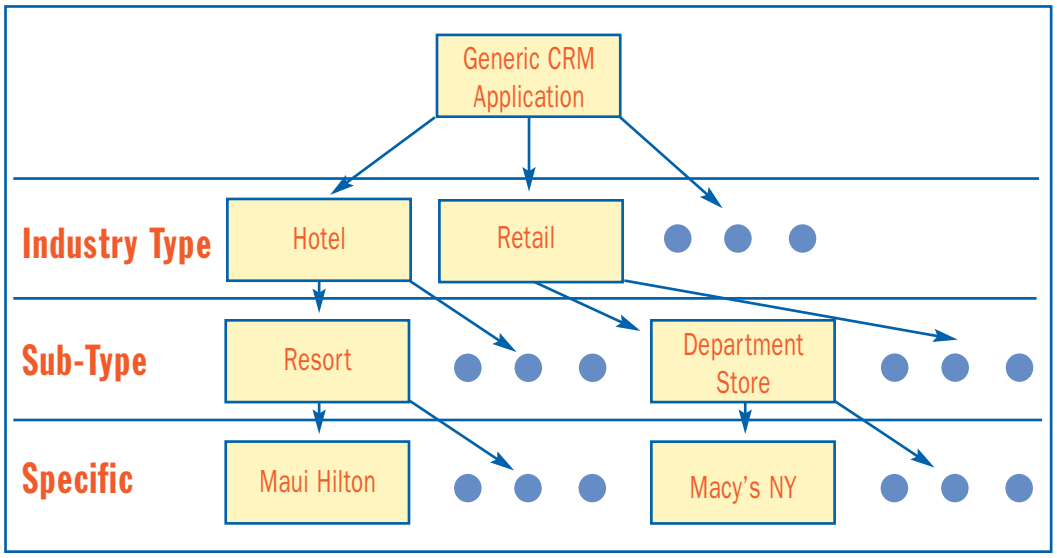

Figure 1: Specializing a CRM application for a particular customer.

So, how does the CRM vendor who is at the top of the food chain have ANY hope of writing a package that is as applicable for Maui Hilton as it is for Macy's New York? Software developers employ several strategies to achieve this, depending on the particular tools they use.

- In a typical RDBMS-based design, there isn't much help available for this. Designers just design the tables with their "best efforts," and hope they did a good job. Each new customer added requires a massive effort to customize the code for the new database design. The vendors typically try to make the best of a bad situation by retaining a professional services team who charge for the customization efforts. Typically, the customization fees equal or exceed the license fees for the product. VARs in each of the sub-industries also handle some of the customization.
- ODBMS-based designs fare a little better in this respect. As new requirements are added or existing requirements modified for each new customer, new objects are added that inherit from, and override, existing attributes and methods. This results in a new schema being auto-generated by the transparent persistence engine of the ODBMS. Thus, the advantage over RDBMS is that the application does not have to be re-designed from scratch. However, massive amounts of new code must still be written. So while the results of customization efforts are better than with the RDBMS, the workload is roughly the same.

In this brief, we'll look at a new approach to object design, based on the Matisse Meta-Class, that enables the development of easily customizable applications.

Meta-Classes & Document-Centric Applications

The example we will be using in this paper is derived from a real live CRM application that has been designed at Matisse Software Inc. Klover, a document-centric application, integrates and classifies full-text documents, rendering them fully indexed and searchable. Some are customer survey documents, either on paper or an e-form. Others are emails from support centers. Still others are "Call Verbatim" documents, which are a transcript of the conversation between a customer and a sales or support representative.

In the rest of this paper, we will examine the problem posed by these polymorphic documents, focusing on the fact that we do not know what new and unique types of documents may be introduced in the future.

Meta-Class vs. Super-Class

Most Object programmers are, of course, familiar with the concept of a Super-Class (which is the class from which the current class inherits). The figure on the left shows the standard UML representation of the Super-Class.

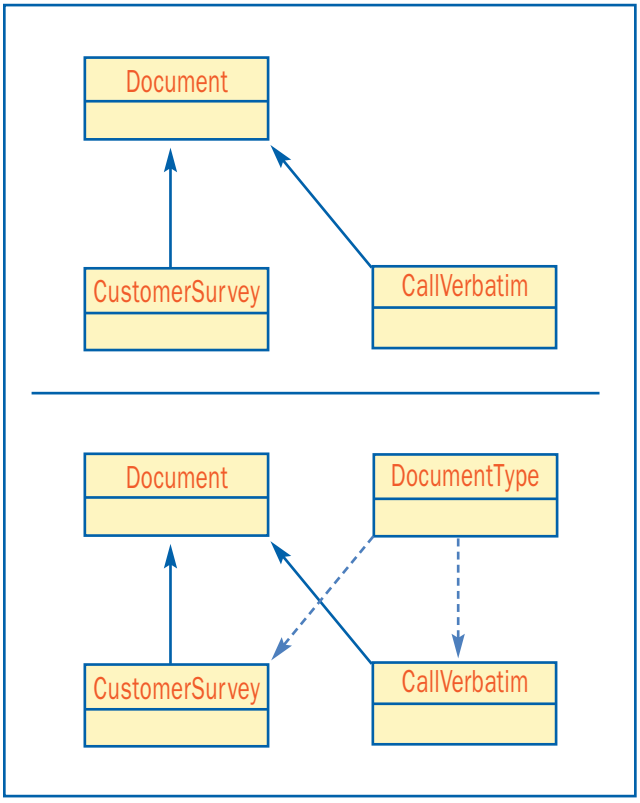

Figure 2: A class can be an instance of another class.

The base class is the Document and the two sub-classes are CustomerSurvey and CallVerbatim…pretty standard OO fare. However, while the concept of a Meta-Class may be familiar to Python programmers, Java programmers may not have encountered it.

Most of us know about objects being instances of a class. What is less familiar is the concept of a class being an instance of another class. When Class B is an instance of Class A, then we say that Class A is a Meta-Class of Class B.

Using the same example, the CustomerSurvey document class and the CallVerbatim document class are now shown as instances of a meta-class DocumentType, in addition to being sub-classes of Document. In UML terminology, the meta-class is sometimes referred to as a Stereotype.

A canonical difference between a super-class and a meta-class is that attributes in the meta-class become static instance variables in the class, and cannot be over-ridden in the class. This is a much more object-oriented way to create static variables. In the same way, methods attached to the meta-class are static final methods in the instance classes.

It is important to note that while the meta-class construct is available in some programming languages (such as Python), and in modeling languages, such as UML, it has never before been available in the context of database design, even among the various ODBMSs. Matisse is unique in making this construct available.

Using Meta-Class to Create Customizable Applications with Matisse

Static Class Variables

One immediate benefit of using a Meta-Class is to have an object oriented way to have static variables in a class. For example, if you need to tag document sub-types as being printable or not, you would add a Boolean isPrintable attribute to DocumentType. Then, CustomerSurvey could have the property set to True and the CallVerbatim could have the property set to False.

Static variables are valuable in creating customizable applications because it gives the programmer a built-in way to classify the classes that may be added later on, and to have the appropriate code work with it correctly. For example, if a new document type is added for a particular customer, and its isPrintable is set to False, the code will not attempt to print it.

Self-Adjusting Methods

While these static variables are cool, the real power behind meta-classing is to use Java Reflection to create truly adaptive code.

Going back to our example, let's say the programmer wants to write a method, which will print ALL the documents in the system, including ALL the printable fields in each docu-

ment, with a template that is specific to the document type (a common requirement). This leads to the following requirements:

- A Static String/VarChar() field associated with each document type, such as CustomerSurvey, that dictates what the overall format of the printed output will look like. This could, for example, store an HTML template to format the document for printing.

- A static property for each text attribute of each of the Document sub-classes that indicates whether this attribute is to be printed, and one for the user-friendly name of the attribute for print purposes. For example, an attribute called custComments may have an isPrintable tag = True, and a friendlyName tag set to Additional Customer Comments.

The power of the meta-class is immediately obvious in this example. We could use a meta-class for the text attributes in addition to the meta-class for the Document classes. Here is what you could do with Matisse:

Notice that we are using meta-classes in two ways: The DocumentType meta-class (which inherits from MtClass) gives us a clean, convenient way to have a static html template string for each document type, and the KTextAtt meta-class (which inherits from MtAttribute) gives us a clean, convenient way to tag each text string with an isPrintable boolean tag, and the friendlyName:String tag.

With this structure in place, we can readily see how the programmer can write a truly adaptive printAllDocuments() method, which does not have to be re-written or over-ridden based on the types of documents that are added to the system for a specific customer. Using Java Reflection, one can discover all the printable fields, and be able to format and print them according to the format specified by the HTML template for the specific document type. If one must alter the application because a new document type has to be added for customers (e.g., CustomerEMail), simply define the printable fields, give a friendly name to each, and create an HTML template for the print output. The code itself remains unchanged.

While it is possible to conceive of a scheme to achieve this goal in the absence of meta-classes, it would most assuredly be more complex, and less straightforward than the method outlined here. For example, one would either require the programmer to remember to define a new static HTML template field for each document type added (because it's not automatically instantiated from a meta-class), or create a new object for the HTML template that has a relationship with the document class. The former is error prone, and the latter adds to the application complexity. As for the text attributes, it gets even messier; one must envision making each text attribute an array with the other attributes attached – again, quite error prone. So while it is always possible to write object-oriented code in a procedural language (like C); it isn't clean, and is much more error prone.

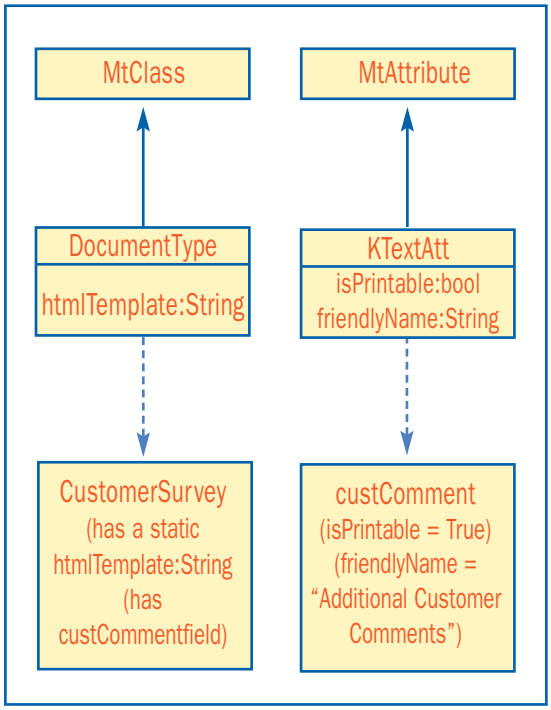

Figure 3: Using metaclasses to write truly adaptive methods.

Conclusion

In this brief we have introduced the reader to the motivation behind the use of meta-classes, and provided a high-level example of how to use them to your advantage in writing applications that can survive the test of time and change.

Matisse Software Inc.
www.matisse.com
433 Airport Blvd, suite 421
Burlingame, CA 94010
650-548-2581