Recognizing the CAICO, A Collection of Almost-Identical Complex Objects

Ted Kaehler

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Memo M-2009-004

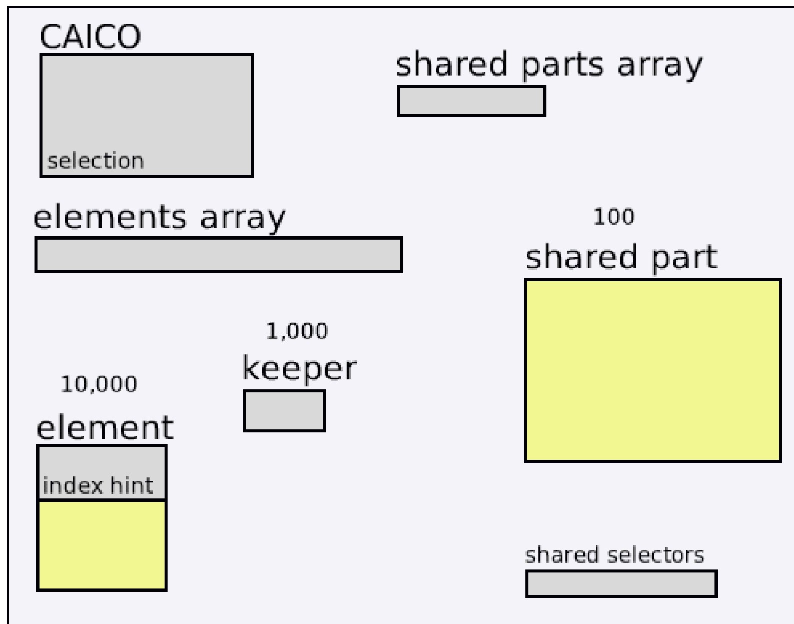
Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

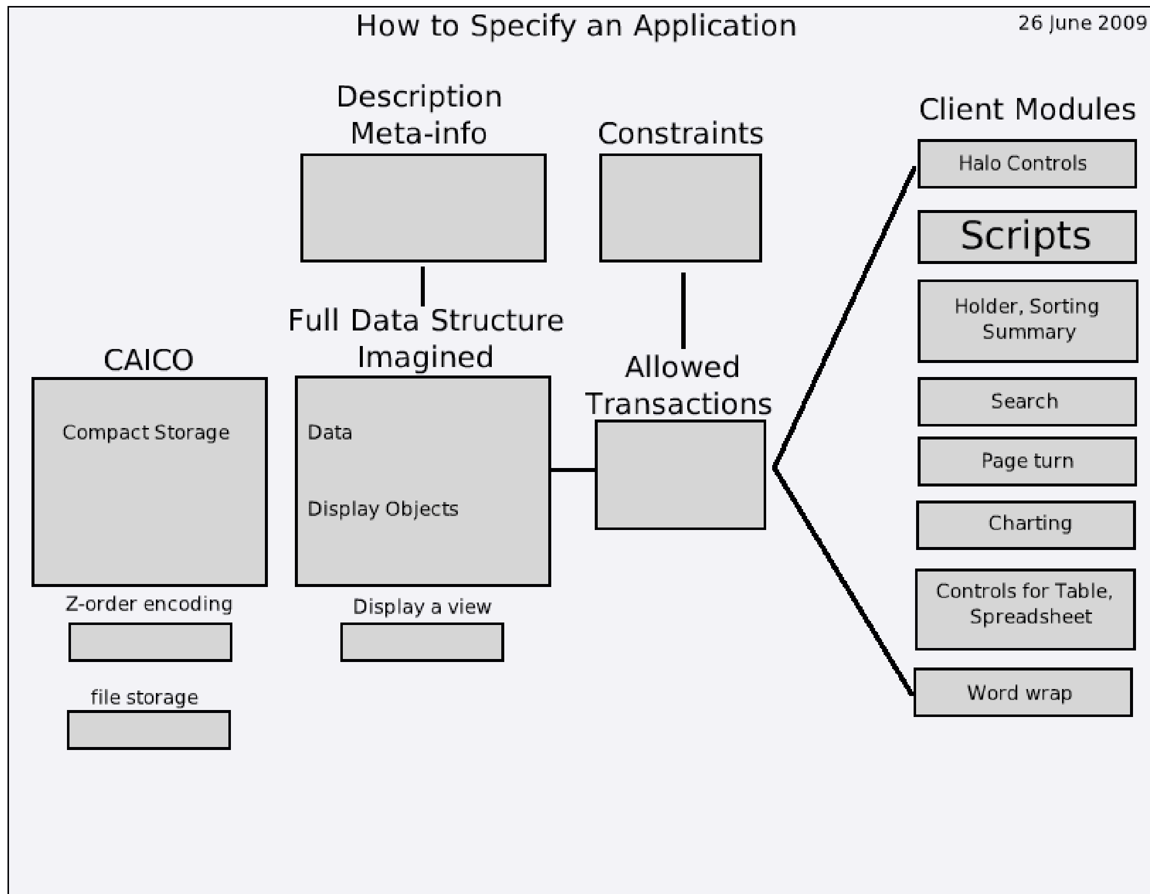
Recognizing the CAICO, A Collection of Almost-Identical Complex Objects

by Ted Kaehler
Feb. 5, 2009

One of the complications of writing a small and elegant text editor is the representation of the text. The cleanest way to work with text is when every glyph is a full-blown object with complete knowledge of all of its properties. As pointed out in "Glyphs: Flyweight Object for User Interfaces' [Ca90], a book chapter with 50,000 complete character objects would consume a lot of space and time. That paper presents an elegant system, but does not propose a general solution to text's data structure problem.

A CAICO, A Collection of Almost-Identical Complex Objects, is a data structure that represents an array of characters and all of their style properties. A user of a CAICO does not need to know the internal representation or the degree compaction via sharing. A CAICO is a classic example of representation-hiding and the "cheat but don't get caught" philosophy.





A CAICO must be able to return any member of the collection as a fully expanded object. It must also be able to return any subset as an array of fully expanded objects, or as another CAICO. When text is passed around the system, it is either in the form of a CAICO or an Array of individual complex character objects.

I would not be making a fuss about this kind of data structure, except for the fact that I have just been dealing with a second example of a CAICO -- a stack of pages in DBJr (a HyperCard-like stack and page system). The model that the user sees in a stack is a collection of pages, each of which is a full-blown nested structure of graphical objects. Most pages use a common background, but sometimes there is a completely different one-page background. If each page had its own copy of the complete graphical hierarchy (as a Squeak BookMorph does), a stack would be huge. Multiple copies of the 'background' objects would take many times the space of the actual data. The special thing about DBJr and HyperCard is that an individual page only needs to store the actual text for that page.

As Yoshiki Ohshima has pointed out, a third example is an array of turtles or patches in Kedama.

The goal is to write down a standard way to create and use a CAICO, so that this kind of collection does not need to be re-invented each time. Here are our two examples:

In Text, each character has a:
character code, font, emphasis, size, color, translucency, visible rectangle (baseline), packing rectangle (width), kern, and baseline elevation (for a superscript). Also, it can find the line in which it appears, its X-Y location and its index in the text. The Text as a whole has a container with a shape and the selection.

In a Stack, each page has:
named text fields, the specific text in this page's fields, graphical objects, page-specific graphical objects, other kinds of parameterized costumes (date, image, numeric field), a background pointer, a stack pointer, a name, a page id, a page number and script variables. The Stack has a name, stack id, and stack script variables. The selection is the current page. It is especially interesting that the instance variables of a page are not fixed, but are added and removed as the user inserts and deletes graphical parts of the page.

In each kind of CAICO, there is a message protocol that each member must understand. In Text, a glyph must be able to display itself on a canvas at a specific place and answer its width and height. In a Stack, a page must be able to produce a complete page costume to be displayed.

A CAICO has a dictionary of the properties. These are the properties of its members, and these properties are handled in several different ways. A dictionary holds specs of the properties, so the CAICO can know what to do when asked. (DBJr has this kind of dictionary for each background.)

Each member has many properties. Some are shared and some are not. To get a property, ask for a member, and then ask the member itself. It is what DBJr uses. The stack returns the actual compact page object, as stored internally, so the request is fast. But that object acts like a proxy for all of its shared properties. For width, it gets the background costume from a class variable, and returns the width. For a named field, it has a getter method installed, which returns the shared field object immediately. For the text on this page, it returns its own instance variable. As properties are added and removed, the Player-like light-weight class of the page adds and removes instance variables, getters and setters. Prototypes in Pepsi do this with ease. The member object only stores what is specific to that instance, but knows how to respond to all properties of the object, shared or not.

For characters to work this way, each "run" of a combination of properties would have a lightweight class or a prototype. A character would be a sibling instance with (class pointer, character code, index in the text). Properties such as boldness are inherited from the prototype.

The only reason we bother with a CAICO is for its compactness. Currently, the programmer must decide what to factor out and share among the instances. For Text, the TextStyle with font and size can be shared, and in a stack the background and field containers are shared. A worthy goal is for the CAICO itself to factor out shared things.

The programmer can give hints. The key is to exploit the uniformity, and not cause any extra work for a programmer who is setting up a CAICO.

Users often want to change many members of a CAICO at once. In DBJr, changing any shared property, such as the location of a background field automatically applies to all pages of that background. This is because the user has the concept of a background in his mind. In text, a user wants to change many characters to bold. The set of objects to change is the selection, or perhaps the entire field. Text, as opposed to a stack, does not have a strong idea of a background. The user does not want to change all characters that happen to be Times 12 and not bold. The answer to this difference is to allow both kinds of ranges for changing a property: the selection or a background.

A very interesting exercise would be to factor DBJr into two parts. One is a generic CAICO to represent the stack and hold all of its parts. The second is all of the specific knowledge and algorithms for being a "stack with pages". That is, everything that distinguishes it from the "text field with characters" example. 'How to turn to the next page' is one of the specific things the DBJr part needs to know.

A CAICO is a legitimate large-scale object, of the kind that we are looking for. It is stronger and larger than a normal object, but not in any way monolithic like a classical application.

[Ca90] Paul Calder, Mark Linton, Glyphs: Flyweight Object for User Interfaces, Proceedings of the ACM Symposium on User Interface Software and Technology, Oct. 1990, pp. 92-101. Third Annual Symposium, UIST, ACM SigGraph, Snowbird Utah, October 3-5, 1990. (Stanford)