At Harlequin created on September 23, 1994 and last modified at Harlequin on August 31, 1995.

Library Design Guidelines

Scott McKay

1. Introduction

The purpose of this paper is to offer some guidelines for doing robust class library design and implementation. In part, this paper is a summary of the paper "Issues in the Design and Specification of Class Libraries", by Kiczales and Lamping (OOPSLA '92).

2. Guidelines for Designing Dylan Components

All Dylan components must obey the following guidelines:

- Provide interoperability with other components on equal terms. That is, Dylan components need to be usable with other components that may or may not be written in Dylan. On the whole, we cannot make the assumption that Dylan components are unique in that they are solely "in charge" of any operating system resources (such as memory, threads, event queues, etc.). We can also not assume that Dylan components will "get control" before other components, since Dylan may be called from other components. We cannot assume that other components will be limited in their use of other library or OS calls.
- Dylan components must not prevent the sharing of operating system resources. We should not make any assumptions that operating system resources might be limited by component boundaries. For example, any component can create a thread which may then execute within other components. Thus, threads created in Dylan must be usable by foreign components and vice-versa.
- It is not acceptable to modify the API to the operating system or standard libraries by "shadowing" the standard calls. Although this can be tempting, it has two major problems: doing so would introduce an interoperability problem for the future; and it's not practical (for example, name resolution in the Windows linking mechanism uses both the symbol and library names, so if you wanted to shadow an API function, you would have to replace the entire kernel library and provide a different version of the masquerading library for each version of Windows — a perturbation that would be visible to all applications, whether or not they used any Dylan code).
- We can implement functionality in Dylan components in any way we want that doesn't violate the constraints above, provided that the Dylan components interact properly with other components via FFI. That is, anything that might be "unusual" about a Dylan component should not be visible on the foreign side of the FFI.

It is a fact that Dylan components will be judged in large part on how well they operate with other components. If the above constraints lead to some loss of efficiency in Dylan components, we must in general simply pay the price, because if an efficient Dylan component does not work properly on an equal footing with non-Dylan components, it will be judged not to work at all.

3. Guidelines for Designing General Classes

The best overall guideline is, make the design just complex enough to solve the problem, but no more complex. Or put another way, design the simplest thing that meets the requirements, but don't make it any simpler.

- Don't think in terms of just classes, think about what protocols each class obeys.
- Don't introduce new classes unnecessarily, since they just add conceptual baggage.

4. Guidelines for Designing Extensible Classes

There are two basic properties we are striving for in class library design:

- Generality, that is, the ability of a library to serve in a wide range of circumstances without modification.
- Extensibility, that is, the ability of a library to be easily modified to meet a particular need.

One other important property for the API for a class library (or any kind of component) is that it should be easy to use. The ease of implementing a component should not be given priority over the component's ease of use.

An issue in the specification of class libraries is, how does one specify an extensible library precisely (so that a client can write specializations or replacement modules), without over-constraining the implementation (which would give the implementor no room to work)? The problem has two major parts:

- The client of a library needs to know that, when they subclass a class, what methods they will inherit.
- The client also needs to know, when they subclass a class, what methods they must define, what methods they can rely on, and what their methods must and must not do.

Our major task in designing for extensibility is to identify what the replaceable parts are in a library and how they interact, and to ensure that other details remain hidden. We need to specify the programmer interface (API) for client users, and also the relationship between parts so that we can maximize extensibility and reuse. We want to allow users to replace parts that have new and different behaviors (that obey the protocols, of course), or have different implementation strategies. All such changes must have predictable consequences.

4.1 Class Graph and Inheritance

The approach of (1) specifying each class in the lattice and its direct superclasses, (2) the set of generic functions, and (3) the set of methods (by giving the generic function and its specializers) is simple, but it's too restrictive (see Kiczales and Lamping for details). In particular, this does not allow for "interposed" classes that might be used to promote code-sharing in the implementation of the library.

To start rectifying the situation, we first define some terminology:

- A *system-defined* definition is one that is part of the library, either specified or implementation-specific.
- A *specified* definition is a system-defined definition that is listed in the library specification, that is, is part of the advertised API of the library. (In Dylan, such a definition will be exported from the library.)
- An *implementation-specific* definition is a system-defined definition that is present in an implementation, but does not appear in the specification. That is, it is not part of the library's API. (In Dylan, such a definition will not be exported from the library.)
- A *user-defined* definition is one that is defined by the user, portable or not. This is the "opposite" of a system-defined definition.
- A *portable* definition is a user-defined definition that depends only on specified definitions. It will work with any implementation of the library.

So:

- An implementor may provide implementation-specific leaf classes as subclasses of the specified classes; such leaf classes can provide methods for any system-defined generic function. (In Dylan, an unadvertised concrete class might be such a leaf class; an instantiable abstract superclass would be advertised in the API.)
- An implementor may provide implementation-specific generic functions on any class, but these definitions should not be visible in user packages/modules.
- Implementation-specific interposed classes are allowed for any specified or portable class C_P that is a subclass of one or more specified classes $C_0 ... C_i$ as long as this condition is met: in the class precedence list of C_p: the classes $C_0 ... C_i$ must appear in exactly the same order as if there was no interposed class. That is, the introduction of an interposed implementation class must not change the ordering of the specified classes in the class precedence list for any specified or portable class.

  It is often the case that these interposed classes will be "mix-in" classes that are used to share implementation rather than define a semantic type. These classes will not typically be instantiable classes. (In Dylan, such classes should not be exported from the library.)
- An implementor may "promote" a method to an interposed superclass only if the method inheritance for the specified generic function stays the same at any specified or portable class, as it would have been with no such promotion.
- User programs must not redefine any specified classes, generic functions, or methods. User-defined methods on specified generic functions must be specialized to a user-defined class (although `eql`-specializers do complicate this a bit).

There is an efficiency issue in extensible classes in Dylan. Classes that are intended to be subclassed must, by definition, not be sealed. However, to gain maximum performance, it is often desirable to seal classes. Clearly, a balance must be struck here. One way to do this is to advertise an abstract, instantiable class that is not sealed, and to have a concrete class that is sealed. Calling **make** on the abstract class creates an instance of the sealed implementation class. When user programs wish to extend the protocol, they must subclass the abstract class rather than the sealed implementation class. To realize the full efficiency gains that can be provided by having sealed classes, the sealed subclass may also have to be exported.

The specification of a Dylan library must explicitly state whether classes and generic functions in the library's API are sealed or open. Furthermore, it must be explicitly stated whether a class is abstract or concrete, when it is instantiable, and when a class is a primary class.

4.2 Protocols

A *protocol* is the behavior given by a set of specified generic functions and methods. In object-oriented class libraries, it is not enough to specify only behavior; one must also specify how each generic function relies on the behavior of other generic functions. That is, the specification of a generic function has two roles:

- It says what each generic function does when it is called, and what each of its methods should do.
- It says how it relies on other generic functions to provides its behavior, in effect providing the "backbone" of the protocol. Specifying this guarantees that all "callee" generic functions will be allowed to fulfill their proper roles.

So the specification for a **pointer-enter** generic function might read:

> This generic function is called by the toolkit whenever the mouse enters the gadget. It arranges for **mouse-inside-button?** to return **#t** until the next call to **pointer-exit** on the gadget. When **pointer-enter** is called, **draw** is also called to highlight the gadget.

It is also important to say something more concrete about the behavior of particular methods. For example, the following specification for the **draw** generic function:

> Draws the gadget on the display. Calls **pointer-within-gadget?** to determine if the gadget should be drawn with highlighting.

might be specialized for a particular push-button containing text:

> For a text button, this draws the button and its text in a 12 point Helvetica font. The button's border is a 2 pixel-thick solid line. The button is highlighted, if necessary, by underlining the text.

4.3 Abstract Classes

Often a library will provide incomplete "abstract" classes that must be subclasses and extended before they can be used. In this case, the specification should state what required methods must be implemented.

Note that there are sometimes methods which should not be overridden. Any such methods should be explicitly noted.

4.4 Efficiency Concerns

Fully specifying the inter-relationships between generic functions gives a lot of power, but it can overconstrain the implementation to the point where efficiency can be lost. For example, specifying that a **drag** generic function must be implemented by repeatedly calling **draw** and **erase** would prevent the implementor from choosing a more efficient strategy, such as drawing into a pixmap and then moving the pixmap. We can use several techniques to help here:

- Use “functional” protocols that allow results to be memoized as long as the memoized results remain valid. The specification describes the conditions under which memoized results remain valid.
- Describe sets of “consistent” generic functions, which are explicitly identified sets of generic functions that behave as though there are calls between them, but are free to in-line knowledge about one another. This means that users who specialize one generic function may need to specialize others as well. The specification of such sets of functions must explicitly identify the relationships between each other.

Note that some generic functions may have “private” pieces of state among themselves (for example, **pointer-enter** and **pointer-exit** may both maintain an “inside” bit). This sort of state may be inferred from the specification, but it is best to state it explicitly.

4.5 Layered Protocols

There is often a tension between ease-of-use and ease-of-extensibility. For example, it is generally hard to write a specification (and the code) for a function that is very powerful, and it is generally easy to write a specification (and the code) for one that is weaker. One way to help here is to layer protocols, by specifying calls to generic functions that are not necessarily visible in the top-level API. For example, the **draw** method for text push-buttons might be specified as follows:

> A text button is displayed as a text string surrounded by a border. The text is drawn in the font given by calling the **font** generic function, and the border is an unfilled rectangle whose thickness is given by calling **border-width**. The button is highlighted, if necessary, by underlining the text.

Such protocol layering allows more code-sharing to take place than might otherwise.

5. Naming Conventions

Here are some suggested naming conventions for module variables:

- Follow the naming conventions given in the DIRM: hyphens between words, angle brackets around class names, asterisks around read/write module variables, a dollar sign in front of read-only variables (constants), question marks at the ends of predicates, and exclamation points at the end of “destructive” functions.
- Avoid abbreviations and acronyms.

- Combine words with hyphens, not just concatenation. Of course, standard English morphology such as prefixes (the **super** in **superclass**) and suffixes (the **es** in **all-superclasses**) is acceptable.
- Combine words with standard patterns. One common pattern is *verb-object*: **add-method**, **freeze-methods**, **remove-duplicates**. Another is *adjective-noun*: **simple-error**, **double-float**, **direct-subclasses**. A third is *object-attribute*: **object-class**, **slot-allocation**. Note that one should not use the pattern *object-verb*: do not say **file-delete**.
- Choose a set of words and stick to them. Don’t use **make** in some symbols and **create** in others for no good reason. Use the words that Dylan provides. For example, Dylan uses the word **slot**, not “field” or “instance variable.”
- Macros that establish some sort of an environment and then evaluate a body in that environment should have a name such as **with-indentation.** If such a macro has a functional interface, then function should be named something like **call-with-indentation** or **do-with-indentation**.
- If you need to introduce a constructor function, it should be called something like **make-point**. One reason for introducing a constructor function is when there are some required initargs; the constructor function should take these as ordinary required arguments. Otherwise, it is best to use **make** <*class*>.
- Instead of introducing coercion functions, specialize Dylan’s **as** operator.
- Accessor (and accessor-like) functions should generally include the name of the thing they are accessing, especially if the name would otherwise be too brief. For example, **text-style-height** and **region-height** instead of just **height**; this is especially important, since the semantics of **height** on a text style is different from its semantics on a region.

6. Other Conventions

The following is a laundry list of other conventions:

- Functions should take their most important arguments first.
- For “copying” functions, the “destination” argument should precede the “source” argument, such as Dylan’s **replace-subsequence!** function.
- “Start” and “end” keyword arguments should be named **start:** and **end:**. The start argument should be inclusive, and the end argument should be exclusive. This is modelled after Dylan’s **fill!** function.

7. Open Issues

We need to provide guidelines for when to seal classes, and when the a sealed subclass of an advertised abstract instantiable class should be sealed. Every library should provide its own guidelines for users about when to specialize on the abstract or the sealed classes (c.f., specializing on **<vector>** *vs.* **<simple-object-vector>**), based on the likely loss of efficiency of a more abstract protocol.

At Harlequin created on September 23, 1994 and last modified at Harlequin on August 31, 1995.