At Harlequin created on September 30, 1994 and last modified at Harlequin on November 9, 1994.

User Interface Design Guidelines

Scott McKay & Chris Fry

1. Introduction

This paper describes the guidelines that programmers should adhere to when designing the user interfaces of DylanWorks tools, and some guidelines for promoting reuse of the code of these tools by separating the user interface from its underlying functionality.

2. General User Interface Design Guidelines

The user interface must make it easy for new users, but at the same time, it should scale up as users learn more, and it must not hinder expert users. In part, this means that all of the basic functionality must be immediately visible so that beginners can start easily. It also means that the simple GUI model is insufficient; the user interface should be a reflection of a deeper underlying language that more expert users can use explicitly to gain more power.

The high level goals we are aiming for in our user interfaces, in no particular order, are:

- Adherence to native look and feel.
- The basic operations of each tool should be immediately obvious, so that users can use a tool successfully right away. There should not be a premium on memorization skills, at least not for beginning users. A related issue is that it should be easy to get in and out of any tool.
- It should always be easy to do common operations. It is less important for it to be easy to do less common operations.
- The interface for a single tool should be self-consistent. All of our tools should be mutually consistent as well.
- It should be obvious how to get on-line help at any point. Note that extensive on-line help is no substitute for overall consistency.
- The tools should not display unnecessary complexity and details. It should be simple to reveal more details as they become interesting.
- There should be a “growth path” that supports becoming more expert. The learning curve should be gentle, but it should not end prematurely. Users should be able to gradually and easily learn how to customize the tools and learn where the short cuts are.
- The interface of any tool should provide an accurate model of what is going on in the application behind the interface. This modelling should allow expert users to access all the functionality of a tool.

- The interface of any tool should provide cues that keep users oriented. This can be done by keeping “landmark” information visible that makes the context obvious, and by providing dynamic feedback (mouse documentation, for example). Interfaces should avoid being overly modal; when there is modality in an interface, it should be visually obvious.
- Design interfaces so that related pieces of information tend to be near each other. It should be easy to navigate from one place to another, especially when pieces of related information are not near each other.
- Each tool should provide immediate feedback. This should obviously be the case when the user has entered a valid command, but it should also be the case when an erroneous action was attempted.
- Error messages should be informative, not mysterious. When possible, error dialogs should present a clear set of choices that describe what actions need to be taken.
- Our tools should generally prevent users from performing an erroneous action. This can be as simple as graying out inappropriate menu items, or as complex as the CLIM model in which all input is dynamically type-checked for validity.
- The communication between tools should be good, but each tool should be free-standing. It should be easy to do multiple tasks at the same time by simply switching tools. When possible, each tool should run in its own thread so that long tasks started in one tool do not prevent the use of other tools.
- The interfaces should be good looking. This means that colors and fonts should be used to good effect to draw the user’s attention and provide constraint; on the other hand, this needs to be done carefully and with restraint. Excessive use of icons should also be avoided. Obviously, the interface should obey “local” platform guidelines.
- In the context of good-looking interfaces, we need to recognize that screen real-estate can be a precious commodity. Careful use of information hiding can help here, but the resulting interface should still be pleasing.
- Text (such as help and error messages) should be grammatically correct, and avoid slang and abbreviations.
- All of ours tools should obey all of these guidelines, so that they have predictable behavior with respect to each other. Having consistent behavior across all the tools means that users do not have to learn idiosyncratic behaviors.

It is the intention that our user interface substrate will make it easy to meet the following goals:

- Adherence to native look and feel.
- Context sensitive help, documentation, and (where feasible) completion everywhere.There should be a single keystroke way of getting context-sensitive help at any point. For exmaple on Windows, this is via the `F1` key. Dialogs should include Help buttons whenever it is appropriate.
- Consistent undo/redo everywhere that remembers more than just the last action. When a destructive operation can’t be undone, there should be a confirmation or a warning.

- Functionality should be available from both the keyboard and the mouse. The input model should scale to support other forms of input, such as voice and tablets.
- Support for generating hardcopy of application-generated output everywhere. (It may prove too difficult to generate hardcopy of everything, such as dialog boxes and entire application frames, because this is often not supported by the native user interface toolkit.)
- Provisions for scripting and interaction logging everywhere. (That is, there should be a way to script anything that can be done from the user interface.)
- Avoid unnecessary mouse motion. In particular, although we should support the traditional select-an-object/select-operation-from-menu paradigm, we should also support more accelerated modes of interaction in which a menu of operations can be gotten directly by clicking on an object.
- Text, such as error messages, should be maintained in an internationalizable way, for example, via text resources.

3. General Guidelines for Implementing Tools

In DylanWorks, we will try to adhere to a general principle that a *tool* is a set of well-defined functionality plus a user interface. That is, when designing and implementing a tool, you should think about what the tool will do, and as a separate task think about what the interface of the tool should be. This has several advantages:

- If the user interface is separated from the functionality, it is simpler to experiment with both the interface and the functionality.
- It becomes easy to have several interfaces for the same functionality. For example, the functionality for browsing classes might be available in both a streamlined graphical browser, and in the editor.
- By separating out the functionality, we are more likely to get “substrate” components that are usable in other areas in unexpected and interesting ways.
- Since the functionality of a tool obeys a well-defined protocol, it is simpler to “compose” tools, such that the output of one tool can serve as the input to another.

4. Intelligent Policies and Defaults

It’s very important that all of the tools, taken together and individually, have a reasonable and self-consistent set of policies. DylanWorks will in no way be policy-free, but many of the policy decisions in DylanWorks will be parameters that can be changed by users. The most important consideration is that we provide a small set of overall policies that we believe are good, and allow users to change policy-sets easily. All of the “factory settings” should implement a good, self-consistent model of use. The “DylanWorks Top-Level Controller” describes this in more detail.