

Cover Page

Job ID: printopdf-36

Title: Getting Ready for CLIM

Requesting User: genera

Cover Page

Getting Ready for CLIM

Preface

In this document, we introduce CLIM with a series of questions and answers that clarify what CLIM is, where it came from, and how Symbolics is involved in it. We then give a more detailed technical overview of CLIM. Finally, we compare and contrast the concepts of CLIM and Dynamic Windows, for the benefit of Symbolics users familiar with Dynamic Windows.

Since the CLIM Specification is not yet complete, some details given in this document might change in the future. The documentation to be provided with Symbolics CLIM will contain more detailed and specific information about converting programs than is available now.

Technical Overview of CLIM

CLIM is an acronym for the Common Lisp Interface Manager. It is a portable, powerful, high-level user interface management system toolkit intended for Common Lisp software developers. The important things to understand about CLIM are:

- *How it helps you achieve a portable user interface* — how it fits into an existing host system; how you can achieve the look and feel of the target host system without implementing it directly, and without using the low-level implementation language of the host system.
- *The power inherent in its presentation model* — the advantages of having the visual representation of an object linked directly to its semantics.
- *The set of high-level programming techniques it provides* — capabilities that enable you to develop a user interface conveniently, including formatted output, graphics, windowing, and commands that are invoked by typing text or clicking a mouse (or “pointer”) button, among other techniques.

CLIM 2.0 does not currently provide any high-level user interface building tools, nor does it provide any sort of high-level graphical or text editing substrate. These are areas into which future releases of CLIM may extend.

CLIM is also not suitable for high performance, very high quality graphics of the sort needed for sophisticated paint, animation, or video postproduction applications. Of course, it is possible to write the bulk of such an application’s user interface using CLIM, and use lower level facilities for drawing in the main “canvas” of the application.

Introduction to CLIM’s Presentation Model

A software application typically needs to interact with the person using it. The user interface is responsible for managing the interaction between the user and the application program. The user interface gets information from the user (com-

mands, which might be entered by typing text or by clicking a mouse), gives that information to the application, and later presents information (the program's results) to the user. Figure 1 shows this common paradigm.

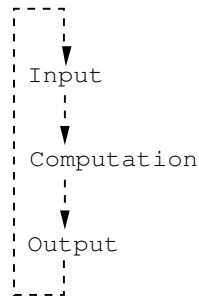


Figure 1. Cycle of Input/Computation/Output

We might describe one conventional approach to the input/computation/output cycle as follows. Invisibly to the user, the application takes the commands and interprets them in terms the program can handle. For example, if the application uses object-oriented techniques, it might build objects based on information garnered from the command and arguments, then manipulate those objects internally, finish its computation, and finally translate from the resulting objects to the appropriate response which is then given to the user. Figure 1 depicts this sequence of events.

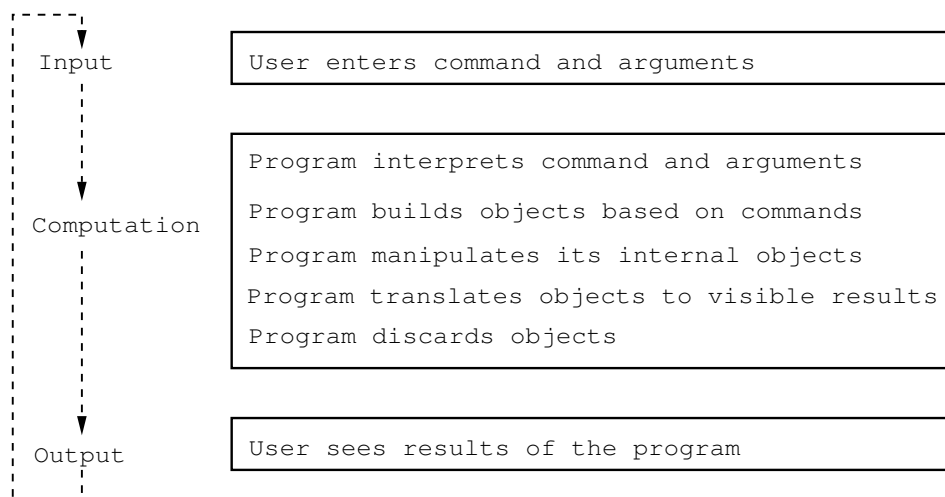
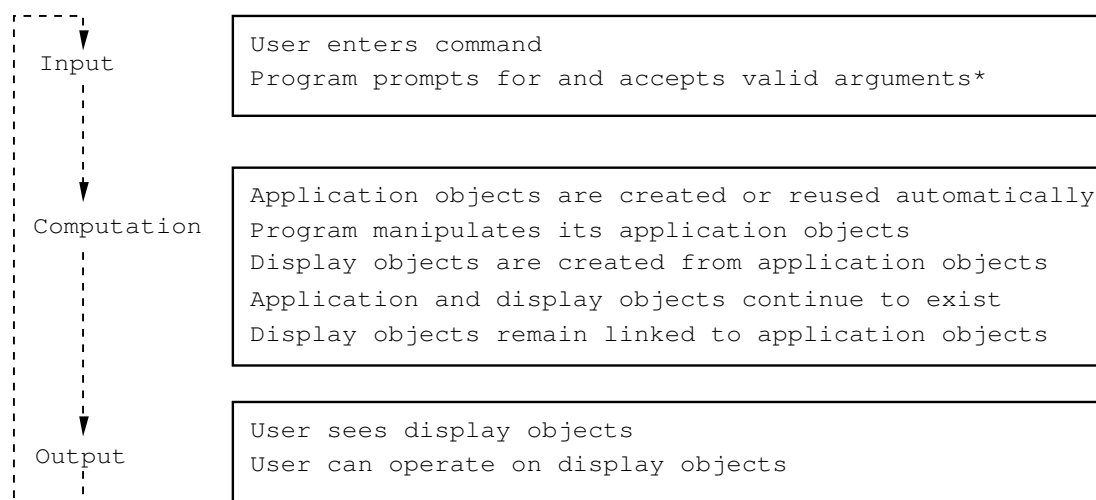


Figure 2. Conventional Approach to the Input/Computation/Output Cycle

The conventional approach uses object-oriented techniques within the computation phase, but the objects do not surface to the user interface. The program performs two translations: from user input to objects, and later from objects to output in-

tended for the user. CLIM revolutionizes the cycle by bringing the power of object-oriented programming to the surface, all the way up to the user interface.

CLIM recognizes that many applications manipulate internal objects which we call *application objects* and they have *display objects*, which are presented to the user. A display object can appear as text or a graphic picture. CLIM supports a direct linking between application objects and display objects. CLIM automatically maintains the association between application objects and display objects, so there is no need for the application to do any translation. Figure 1 shows how CLIM views the cycle of input/computation/output.



*The command and arguments can also come from a single gesture.

Figure 3. CLIM Approach to the Input/Computation/Output Cycle

In effect, CLIM replaces some of the tedious and error-prone steps of the conventional user-interface model with higher-level object-oriented techniques. The advantages of the object-oriented user interface are subtle but extremely powerful (in fact, you might not recognize them at first glance, but they will grow on you gradually as you develop your CLIM applications):

- A command is structured so that the user interface understands something of the semantics of its arguments. That is, each argument must be an object of a specified type. This helps the user in several ways:
 - The user is prevented from entering invalid input, because the user interface automatically enforces the validity of each argument.

- The user can get online help or prompting from the user interface, based on the type of the argument.
 - The user can enter input in creative and convenient ways, such as by clicking on object displayed on the screen by a previous command. The user interface knows which displayed objects are valid within the current context, and can make them *sensitive* (the objects are highlighted as the pointer passes over them).
 - The user has a flexible means of interacting with the application, and often can choose whether to use the mouse or keyboard to communicate with the application.
- In CLIM, the user interface directly reflects the application's structure, because the display objects stand for application objects. Unlike the conventional model, a CLIM user interface is not tacked on the application as a separate entity which can diverge from the application to ill effect. CLIM's direct linking between the application and display objects ensures a natural consistency between the application and its user interface.
 - Display objects are organized in a type lattice in the usual object-oriented way, so inheritance can be used to good advantage. For example, when the user is entering an argument of a given type, objects of that type *and its subtypes* are valid as input. For example, an application might define display objects representing buildings, schools, and houses. When the application needs a building as input, the user can enter a school, because school is a subtype of building. CLIM also offers a library of predefined types, saving the application programmer some effort when dealing with commonly used display types.
 - Objects can be shared freely among different applications. CLIM's ability to share objects directly contrasts to some conventional systems, in which data can be shared among applications only by reducing it to its lowest common denominator (usually text).

How CLIM Helps You Achieve a Portable User Interface

CLIM provides a consistent stream-oriented interface to window systems across a large set of hosts. When developing a portable user interface, you write your application in terms of CLIM windows and their operations. You can also use Common Lisp and CLOS. Figure ! shows the elements on which your application depends.

Your application is portable because it depends only on languages which have been standardized: Common Lisp, CLOS, and CLIM. Of course, porting is never entirely effortless, because different implementations of standardized languages can differ from one another in minor ways.

From the perspective of your application, the details of the host window system, host operating system, and host computer should be invisible. CLIM handles the

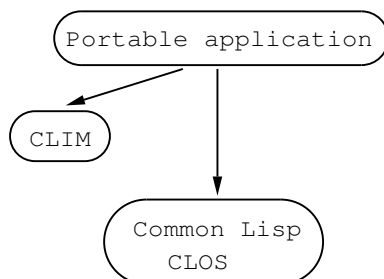


Figure 4. Foundation of a Portable Application

interaction with the underlying window system. Figure 5 shows the elements of the host system from which CLIM shields your application.

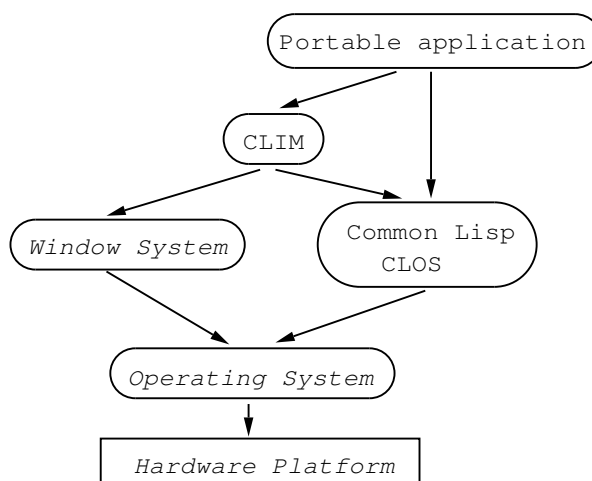


Figure 5. How CLIM is Layered Over the Host System

CLIM shields you from the details of any one window system by abstracting out the concepts that many window systems have in common. Using CLIM, you specify the appearance of your application's output in general, high-level terms. CLIM turns your high-level description into the appropriate appearance for a given window system.

In some cases, you might prefer to control the appearance of your user interface more directly. You can bypass CLIM and use functions provided in the underlying window system or toolkit to achieve more explicit control, at the expense of portability.

Highlights of CLIM Tools and Techniques

CLIM offers the following tools and techniques:

Windows and events

A portable layer for implementing *sheet* classes (types of window-like objects) that are suited to support particular high level facilities or interfaces. The windowing module of CLIM defines a uniform interface for creating and managing hierarchies of these objects regardless of their sheet class. This layer also provides event management.

Graphics

A rich set of drawing operations, including complex geometric shapes, a wide variety of drawing options (such as line thickness), and a sophisticated color inking model. CLIM provides full affine transforms, so that you can perform arbitrary translations, rotations, and scaling of drawings.

Output recording

A facility for capturing all output done to a stream, which provides the automatic support for scrollable windows. In many cases, programs produce output in the form of *display objects*, which can be manipulated directly by the user (see context-sensitive input). Thus, not only is the output recorded — whether textual or graphic — but it also retains its semantics and can be reused when appropriate.

Formatted output

High-level macros that enable you to produce neatly formatted tabular and graphical displays with minimal effort.

Context-sensitive input

Simple, direct means of using a displayed output object as input. As mentioned above, an application can produce output via objects, which retain their semantics. Users can recycle visible output into input for the same application or a different one. Each CLIM application sets up a context for what kind of input is expected at a given time. For example, a CAD program that supports designing electrical circuits might have a command called *Set Resistance* which sets up an input context in which a resistor object is expected. Any resistors appearing in the CAD programs display are automatically made mouse-sensitive in that context, so the user can click on one to enter it as input.

Adaptive toolkit

A uniform interface to the standard compositional toolkits available in many environments. CLIM defines abstract panes that are analogous to the gadgets or widgets of a toolkit like Motif or OpenLook. CLIM fosters look and feel independence by specifying the interface of these abstract panes in terms of their function and not in terms of the details of their appearance or operation. If an application uses these interfaces, its user interface will adapt to use whatever toolkit is available in

the host environment. By using this facility, application programmers can easily construct applications that will automatically conform to a variety of user interface standards. In addition, a portable CLIM-based implementation of the abstract panes is provided.

Application-building facilities

High-level facilities for defining applications, helping you to lay out windows and gadgets, manage command menus and menu bars, and link user interface gestures (such as mouse clicks) with application operations. The application-building tools help you construct a flexible user interface that can grow from the prototype to the delivery phase.

Comparing and Contrasting DW and CLIM

This section describes CLIM in terms of how it is similar to and how it differs from Dynamic Windows. It also discusses conversion issues.

CLIM offers some advantages over Dynamic Windows. In brief, these are:

- Ability to develop a portable user interface.
- Support for using toolkits offered on various window systems to achieve the look-and-feel of a given system.
- Simplification of some Dynamic Windows functionality, resulting in greater ease of use and superior performance.
- Exposed underlying protocols, enabling you to modify or extend the behavior of CLIM.

Converting an Application From DW to CLIM

Genera users do not have to convert programs from Dynamic Windows to CLIM. Symbolics will continue to support DW in Genera. In fact, it is possible that the portions of Genera that use Dynamic Windows will not be reimplemented in CLIM.

One good reason to convert an existing program to CLIM is to take advantage of the portability benefit that CLIM provides. If your goal is to deliver an application with a user interface on a variety of Lisp platforms with different window systems, you will probably want to convert the application's user interface to CLIM.

Another good reason to convert a Dynamic Windows program to CLIM is that many of the high level facilities in CLIM are faster than in Dynamic Windows.

Symbolics provides a conversion tool to help automate the procedure of converting programs from DW to CLIM. For more information, see the section "Converting from Dynamic Windows to CLIM".

When developing a new application, you will need to decide whether the user interface should be programmed in Dynamic Windows or CLIM. Although both will

coexist in Genera, there is no direct compatibility between them, and hence no mixed programming approach.

Converting an Application From TV to CLIM

Some Genera users have applications that depend on Release 6 window functions in the **tv** package. In some cases, these applications were not converted to Dynamic Windows because of performance reasons. CLIM's performance is superior to that of Dynamic Windows, so for performance reasons, users may want to convert programs that use Release 6 window functions to CLIM.

Note that the Release 6 window system has a very different architecture from DW or CLIM. For example, window programs typically define new flavors of the window with methods that handle very low-level events (such as refresh and mouse motion). Because of this architectural difference, converting from **tv** to CLIM usually requires a careful redesign of an application's user interface, and the usefulness of automatic conversion tools is limited.

Similarities Between Dynamic Windows and CLIM

- CLIM supports essentially the same presentation model as that in Dynamic Windows. CLIM captures the Dynamic Windows' philosophy that a program's user interface should reflect its semantics.
- CLIM provides a graphics model which is similar to that of Dynamic Windows, but is simplified and more uniform.
- CLIM includes an application-building tool similar to the **dw:define-program-framework** of Dynamic Windows.
- CLIM's command processor is virtually identical to that of Dynamic Windows.
- CLIM's input editor is a simplification of that of Dynamic Windows.
- CLIM supports a version of Genera's character styles. In Dynamic Windows characters, strings, and displayed text have style. In CLIM only displayed text has style.
- CLIM supports completion and context-sensitive help in the spirit of Dynamic Windows.

CLIM as a UIMS, and Not a Window System

Dynamic Windows plays two distinct roles. It is both a window system and a user interface management system. CLIM is not a window system; it is layered on top of some other window system, such as X, NeWS, or Microsoft Windows. Therefore, CLIM recasts the interfaces of Dynamic Windows related to being a window system in a portable manner. CLIM encompasses the functionality of many window systems; it acts as an "abstract window system" or a "generic window system" which can be layered on top of another window system. CLIM enables you to develop a portable user interface, whereas Dynamic Windows does not.

The portions of Dynamic Windows that are directly related to its role as a window system are not included in CLIM. For example, in Dynamic Windows, you can operate on a window by sending it a message because some dynamic windows are implemented as flavors that use the message-sending paradigm. CLIM does not support that paradigm.

CLIM, like the second role of Dynamic Windows, is a user interface management system. CLIM shares the philosophy that you as programmer should be able to express what you want to do in high-level terms, and the system should manage the details for you.

CLIM is Built up From Layered Protocols

Whereas Dynamic Windows includes a great deal of flexibility in its single documented interface, CLIM is a layered protocol in the spirit of CLOS. In this document, we refer to the higher level as the CLIM Application Programmer Interface (or API) and the underlying level as the CLIM Class Protocol.

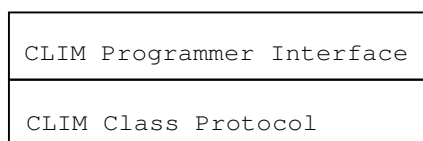


Figure 6. CLIM Protocol Layers

At the API level, an important design goal is that there should be one simple way to do something. There can be some exceptions to this goal; when a very common idiom is identified, it might be included even if there is another (more verbose) way to do the same thing. Where some Dynamic Windows functions and macros offer many keyword arguments, CLIM pares these down to a minimal set without sacrificing functionality.

The CLIM Class Protocol is exposed to allow advanced users to modify or extend CLIM in the object-oriented way. The API functions and macros are implemented in terms of the CLIM Class Protocol. The CLIM Class Protocol, for the most part, is not documented in this book. If you are interested in the CLIM Class Protocol, you should consult the **CLIM II Specification**.

Comparing the Presentation Type Systems

Dynamic Windows allows the presentation type lattice to be computed at run-time. In Dynamic Windows, using inheritance can get complicated, because you must specify what happens at run-time. In CLIM, the type lattice is fixed at load-time, as it is in CLOS. By fixing the type lattice at load-time, CLIM achieves a performance improvement and simplifies the conceptual model. In practice, this restriction has had no negative effects on any applications, and has the benefit of making

CLIM's presentation type system far faster than the Dynamic Windows presentation type system.

The CLIM presentation type system is a straightforward extension of the CLOS type system. In CLIM, defining a presentation type is similar to defining a CLOS class. CLIM extends the CLOS type system by supporting parameterized types, such as integer ranges. This has the benefit of making the CLIM presentation type system "feel" almost exactly like CLOS.

CLIM's Unified Geometric Model

CLIM includes a unified geometric model which is used to represent windows, graphics, and widgets. In other words, everything from a window itself to the graphics drawn on it conforms to the same geometric model. This enables you to deal with windows and graphics in a uniform way. CLIM also provides a general model for transforming, rotating, and scaling geometric objects. CLIM's unified geometric model results in a simplification of some mechanisms used in Dynamic Windows.

CLIM and User Interface Appearance

It is an ambitious goal of CLIM to bridge a wide gap between two styles of user interface programming.

In Genera's style, the principal goal is for the user interface to convey the application's semantics. This goal leads to a natural consistency between the application and its user interface. However, Genera and Dynamic Windows have been weak in enabling programmers to specify a unique and attractive appearance of the user interface. In other words, Genera has tended to sacrifice form for content.

Many commercial toolkits have powerful means of controlling the visual appearance of a user interface. Traditionally, these toolkits offer no support at all for connecting the application's semantics to its user interface. The user interface is thus designed and implemented as a separate, add-on piece to the application. In other words, the toolkits tend to sacrifice content for form.

What's missing from each of these approaches is the connection between the semantics and the appearance of a user interface. CLIM enables the programmer to specify the semantics and appearance of the user interface in an integrated way. It provides the glue between the two.

For example, suppose that your user interface wants to use a dialog to read a real number in the range from zero to ten from the user. A conventional toolkit might make it easy to provide a visually attractive slider to prompt the user, but when the application receives the input, there are no semantics associated with it; the programmer must write some callback that handles events on the slider and converts them to the desired real number. In Genera, the straightforward way to get the number is to give a textual prompt such as "Enter a number from 0 to 10". The appearance of the prompt is not particularly appealing, but when the input arrives, Genera knows its semantics; it is a real number in the correct range.

CLIM aims to include the strength of each of these paradigms. The presentation model maintains the link between the application's semantics and its user interface. The adaptive toolkit enables you to provide a visually attractive user interface. So, if you want to use a slider to get a real number in the range from 0 to 10 from the user, you can use the following:

```
(clim:accept '((real 0 10)) :view clim:+slider-view+)
```