

# The “Chalk” User Interface Architecture for the Slate Environment

Brian T. Rice

8th August 2004

## Contents

### 1 Overview

The Chalk system for the Slate environment is a user-interface architecture that combines the experience and ideas of many different systems in a new and comprehensive manner. It provides integration and well-conceived factorings of many user interface concepts that make it ideal for both an application-development environment as well as an end-user environment. It also provides a flexibility in being adaptable to different display types (terminals and host environments), and a natural extensibility in providing a cooperative open environment where applications benefit each other and combine functionality without undo effort or complication.

### 2 Design Sources

#### 2.1 Morphic

##### 2.1.1 Overview

The Morphic user interface architecture was introduced in the Self system, and later adopted and expanded within Squeak Smalltalk. It makes a lot of considerations for end-user programming, and providing a certain amount of power ahead of the developer’s usual interaction in an array of views on programs’ source text.

##### 2.1.2 Principles

**Direct manipulation** With morphs, “What you see is what it is.” They can double as pieces of application functionality as well as the graphical elements that represent it. Much effort is placed in supporting this idea with development tools which are available through direct interaction with a morph, mostly

through some sort of interactive inspector and editor to modify attributes and behavior.

**Self-Contained Objects** While Morphic screen elements often have domain model objects driving them, it is nevertheless the core concept that morphs can have their own state and behavior, and can be picked up and dropped into another environment and work in much the same way. Animation is even provided through a single hook method to self-update. Generally, this suggests a prototype-style object system, with methods definable on any object and cloning of objects being the simplest version of duplicating behavior. The Squeak implementation manages to achieve this to a decent degree by automating the provision of these features over a class-based system.

### 2.1.3 Concepts

**Morph** A display object which has both behavior and a variable amount of state.

**World** The top-level display object which controls the coordinate system, deals with the underlying drawing context, and is a morph as well.

**Step** The animation and updating protocol method. Each world propagates a periodic stepping message to its underlying elements recursively, which act on it as they see fit.

**Dependents** The mechanism for notifying an unforeseeable number of interested third-party objects of state changes to a certain object.

**Cloning** New graphical elements do not have to be generated from a factory method, but can instead be copied from a template or from a live, working example.

### 2.1.4 Components

Outliners, halos, menus, workspaces...

### 2.1.5 Language Semantics

Self - object-oriented, prototypes, multiple dynamic inheritance, single dispatch.

Squeak Smalltalk - class-based, single inheritance, tries to emulate Self features in order to make working with morphs livable.

## 2.2 CLIM/DUIM

### 2.2.1 Overview

CLIM stands for “Common Lisp Interface Manager”. The term “interface manager” reflects the abstraction with which CLIM deals with interface issues; generally it invites creating applications as an extension of a general framework

and substrate, and also as introducing a service within a system. Moreover, it reflects a philosophy of system design in that CLIM takes over the entire input/output cycle of an application environment.

CLIM began as a descendant of the Dynamic Windows graphical interface manager, developed at great cost for Symbolics' Genera platform running on dedicated, high-end hardware. As a result, this system developed with little concern for wide-spread usage: CLIM was merely developed to make a portable variant, even with a provisional specification for vendors to follow. The DUM toolkit is the "Dylan User Interface Manager", representing an adaptation of CLIM to modern graphics resources, while also simplifying and rationalizing much of the library interfaces.

### 2.2.2 Principles

Abstraction from platform-specific issues: displays, input methods. Windowing systems are treated as back-ends, along with the widgets they provide.

A generic framework for nouns (presentation types), and verbs (commands and presentation methods).

Interfaces are generated in a semi-automated fashion, through this abstraction.

### 2.2.3 Concepts

**Region** The various types of geometrical shapes on a screen. These types separate out a lot of the display occlusion and composition logic into a rational set of operations.

**Design** Regions generalized to encompass composable patterns. This is an area where historically multiple inheritance has been leveraged to a huge degree to provide a lot of genericity.

**Input Record** An input stream - a provider of input to the various screen areas, whose elements could be cached and replayed.

**Output Record** A kind of display list - a collection of instructions for drawing something on the screen. Output records can contain other output records as well, forming a display tree. They also can act as a drawing history, which can then be partially or totally replayed or used for scrolling and layout re-use.

**Presentation** Associates a type of object, an instance of that type of object, and that object's visual representation. Basically, when something should be represented, and output record is created associated with the object in question and the type of presentation governing the appearance and use of it.

**Presentation Type** A user-interface data type, which describes in one place all the information about an object necessary to display it to the user and

interact with the user for object input. Sometimes these types are related to the underlying program objects *as such*, but this is not necessarily the case.

**Translator** A translator is a function which can dynamically provide a relevant meaning for a presentation type outside of the context it was intended for.

**Command** Represents a single user interaction. It has a symbolic name as well as an argument signature. Historically, these have been mapped to menu items and shell commands.

**Present/Accept** The main input/output pairing associated with the presentation concept and translations. Presenting basically takes some object, renders it on screen, and tags the on-screen representation with the object and presentation type. Accepting is a system for input which can draw from the available on-screen presentations. A command may request an argument of a certain type. This alters the context to a situation where objects which satisfy this type of input are sensitive to be used for input on a single mouse-click. Mouse-overs will query display objects and they will highlight themselves to indicate sensitivity in the context. Also, a system of name completion is also provided within input fields where text is the mode, and the scope is also context-sensitive in the above sense.

**Gesture** A single input event, from various kinds of devices, such as mice, keyboards, digitizers, or other kinds of tablets or touch-screens.

**Medium** Handles the device-specific information for drawing objects. Basically there is one medium for every underlying graphics architecture or context used.

**Sheet** Specifies the destination for the graphical output of a medium, while being portable. Something of an ideal graphics plane, which is rendered down to the non-ideal medium.

**Pane**

**Port**

**Gadget** Represents a widget or control or decoration in an underlying graphics system. These are abstract, so that a certain widget will be implemented on various platforms in the native way, or also so that CLIM or the application can provide a substitute where the underlying graphics system does not implement it.

#### 2.2.4 Components

The listener, the menus, the status line, the application, the command tables...

### 2.2.5 Language Semantics

CLOS / Dylan - object-oriented, classes, multiple dispatch, singleton-specializers, macros, closures, optional keywords.

## 2.3 Others

### 2.3.1 Garnet

constraints.

### 2.3.2 Cells

simpler constraints.

### 2.3.3 Fabrik

pipe-style event processing.

### 2.3.4 Merlin

2-d/3-d integration

## 3 Goals

Painless interoperation of user-interaction programming and large-scale application interface development.

## 4 Architecture

### 4.1 Blitting

### 4.2 Geometry

The geometry concepts are straightforwardly adopted from CLIM/DUIM.

#### 4.2.1 Types

**Region** A geometrical space with a notion of inside vs. outside.

**BoundRegion** A region which has bounded extent.

**Everywhere** The region encompassing all others.

**Nowhere** The empty region.

**RegionSet** A set of regions.

**RegionUnion** A RegionSet that acts as a union of those regions.

**RegionIntersection** A `RegionSet` that acts as an intersection of those regions.

**RegionComplement** The “outside” of a `Region` as a `Region`.

**Point** A 0-dimensional `Region` at one location.

**Area** Any 2-dimensional region.

**Rectangle** A rectangle-shaped `Area`, aligned with coordinate directions for simplicity.

**Trace** Any 1-dimensional `Region`.

**Path** A simple 1-dimensional series of `Points`.

**LineSegment** A portion of a straight line, defined between two `Points`.

**Polygon** A (possibly) closed series of `LineSegments`. If it is closed, there is a notion of interior for it.

## 4.2.2 Operations

### Generic Region operations

`\|` Union; successive unions add to a single collecting `RegionUnion`.

`\&` Intersection; successive intersections add to a single collecting `RegionIntersection`.

`-` Geometrical difference.

`complement` Returns a `RegionComplement` corresponding to the argument.

`contains:` Compares regions to see if one contains all of the other.

`intersects:` Compares regions to see if any area is in common between them.

`boundingBox` Returns a (smallest) `Rectangle` which contains the given `BoundRegion`.

## 4.3 Scene Structure

### 4.3.1 Overview

### 4.3.2 Types

### 4.3.3 Operations

**SceneElement** An `OrderedTree` object specialized to with display-specific knowledge.

#### 4.3.4 Operations

### 4.4 Drawing

#### 4.4.1 Types

**Canvas** An abstraction over drawing targets or rendering backends. It is a surface to be drawn *upon*, with state and updatability.

**Medium** An abstraction over stylistic drawing options, such as the color and pattern of the ink, or thickness / shape of the pen used to draw shapes. Each canvas has a medium attached to it describing its default options, allowing the user to ignore such details for the most part. The user may also supply a more specific medium to draw with, composing over the default.

**Design** A set of drawing options or effects that can be combined *before* rendering with other designs or regions to make a particular visual effect.

#### 4.4.2 Operations

### 4.5 Layout

#### 4.5.1 Space Requirements

#### 4.5.2 Layout types

### 4.6 Transformations

#### 4.6.1 Types

#### 4.6.2 Operations

### 4.7 Input/Output and Recording

#### 4.7.1 Events

#### 4.7.2 Aggregate Events

#### 4.7.3 Event Processing

#### 4.7.4 Recording and Replaying

### 4.8 Presentations

#### 4.8.1 Overview

Presentations are domain-specific user interface data types. Each application has its own set of semantically significant user interface entities; a CAD program for designing circuits has its various kinds of components (gates, resistors, and so on), while a database manager has its relations and field types. These entities have to be displayed to the user (possibly in more than one displayed representation) and the user has to be able to interact with and specify the

entities via input actions. Frequently, each user interface entity has a corresponding Slate object type (such as an application-specific structure or Slate object's definition), but this is not always the case. The data representation for an interaction entity may be a primitive Slate object type. In fact, it is possible for several different user interface entities to use the same kind of Slate object for their internal representation, for example, building floor numbers and employee vacation day totals could both be represented internally as integers.

Chalk provides a framework for defining the appearance and behavior of these user interface entities via the `PRESENTATION TYPE` mechanism. A presentation type can be thought of as a Slate object that has some additional functionality pertaining to its roles in the user interface of an application. By defining a presentation type the application programmer defines all of the user interface components of the entity:

- Its displayed representation, textual or graphical.
- Textual representation, for user input via the keyboard.
- Pointer sensitivity, for user input via the pointer.

In other words, by defining a presentation type, the application programmer describes in one place all the information about an object necessary to display it to the user and interact with the user for object input.

The set of presentation types forms an inheritance heterarchy, an extension of the Slate object heterarchy. When a new presentation type is defined as a sub-type of another presentation type it inherits all the attributes of the supertype except those explicitly overridden in the definition.

#### 4.8.2 Manifest Presentations

Manifest presentations are those constructed by an explicit tagging by the parent application on its output elements with some type and formatting. These descend from CLIM's presentation notion.

#### 4.8.3 Latent Presentations

Latent presentations are those that exist because the nature or structure of the output itself. For clarity's sake, latent presentations only become available on demand due to the context, without an explicit tagging required *a priori*. This is the primary advantage of this type of presentation usage, in that application design may be extended at run-time to support vocabulary and usage by other applications.

#### Examples

1. Looking up a word in a language dictionary, since any text on the screen can contain what amount to applicable content for, say, a command which is the interface to this functionality. So, selecting this command puts the

context in position to accept from a stream anything which dynamically parses as a word.

2. Basing a text editor's actions and selectivity on the latent presentation types of characters, words, phrases, sentences, and paragraphs. In a similar manner as the Emacs editor architecture, content could be notated to have a certain mode, which defines the effective meaning of these implicit presentation types and the actions upon them within a certain view on some content.
3. For editing graphics, a mode of operation may be desired which applies to anything with a region on the screen. In this case, the regions of the visual elements would be queried for their geometrical type, and the available operations for this latent type could be based upon the provisions of those for a graphical editor application which uses manifest tagging for its own operation.

**Built-in vs. Plug-in Support** There are two cases of support for presentation types in the latent presentation case. The first is where the application itself is made with some knowledge of this presentation type, or perhaps has defined it on its own. In this case, it is desirable to allow the application to determine the method for accepting them within the context. Specifically, when a presentation query is performed on a visual element, an upward search in the visual graph of elements should query application objects that it encounters for supported types. Types matching the current context's acceptance parameter would then have their accept methods run over the bottom-level scene elements, and the innermost-found object would be the last searched-for.

The "plug-in" or third-party support for presentation types would have lower precedence than the means an application itself provided for any types. The natural way to fit this in to the previous scheme is to define support for them in the applicable world, corresponding to a top-level scene graph element.

This scheme allows for context-dependent treatment of types and their parse, through the precedence toward the innermost support.

#### 4.8.4 Selections

The act of selecting and dynamically grouping on-screen elements could be enhanced greatly by providing a menu or action context for it which is built by applying some ACCEPT method accepting based on installed presentation types; this would provide another form of accessing latent presentations, differentiated in that the selection provides a manual hint of the scope. Such ACCEPT methods for usability's sake should be ordered for precedence in terms of the desirability of certain presentation types over others by the user in a certain context, as well as limited by a certain threshold: beyond a certain point, too many distinctly parsed possibilities would overwhelm the user, and some will be beyond any interest level, unless no other possibilities are valid.

#### 4.8.5 Presentation Types

**Presentation** A `SceneElement` annotated with an associated presented object and the semantic type used with that object.

**View** An object representing a kind of output format style for a presentation to use. This is usually associated with an output medium, but may be mixed via user preferences or various constraints of necessity (like brevity in a complicated display collection).

#### 4.8.6 Accepting Methods

#### 4.8.7 Presentation Methods

### 4.9 Commands

### 4.10 Control-Flow

#### 4.10.1 Multi-Threading Interactions

#### 4.10.2 Continuations

### 4.11 Morphs

#### 4.11.1 Concept

Latent presentations form part of the basic notion of a MORPH in that the interface element can be considered as a thing-in-itself for operational actions and queries upon them. In the sense that latent presentation could be provided within a generic query mechanism on the on-screen elements, it is more general by being more “late-bound” than the traditional morph concept. In this way, the morph concept is reduced to a more fundamental idea that the interface elements on the display may have state and behavior in and of themselves, or “as they are” instead of “as they were created”.

#### 4.11.2 Basic Manipulations

#### 4.11.3 Outliners

#### 4.11.4 Visual Tools for Visual Elements

## 5 Development Style

How to build functionality and applications in Slate - the various methods.

## 5.1 Direct Manipulation

### 5.1.1 Typing “on air”

### 5.1.2 Typing to an object

### 5.1.3 Using Outliners

## 5.2 Vocabulary Definition and Display

### 5.2.1 Defining Presentation Types

### 5.2.2 Defining Presentation Methods

### 5.2.3 Installing Support into Applications

## 5.3 Combinations

# 6 Applications

## References

- [1] *Programming as an Experience: The Inspiration for Self*. Randall B. Smith and David Ungar. Sun Microsystems Laboratories. ECOOP Proceedings, 1995.
- [2] CLIM 2.0 Specification
- [Kras88] *A Cookbook for Using the Model-View- Controller User Interface Paradigm in Smalltalk-80*. G. E. Krasner and S. T. Pope. JOOP, vol 1, no 3, August/ September, 1988, pp 26-49.
- [Deac95] *Model-View-Controller (MVC) Architecture*. John Deacon. 1995. Available Online <http://www.jdl.co.uk/briefings/>
- [Myer96] *Re-usable Hierarchical Command Objects*. Brad A. Myers and David S. Kosbie. Proceedings CHI'96: Human Factors in Computing Systems Vancouver, BC, Canada. April 14-18, 1996. Available Online in HTML [http://www.acm.org/sigchi/chi96/proceedings/papers/Myers/bam\\_com.htm](http://www.acm.org/sigchi/chi96/proceedings/papers/Myers/bam_com.htm) in PostScript <ftp://ftp.cs.cmu.edu/afs/cs.cmu.edu/project/amulet/www/papers/commandsCHI.ps>.
- [3] Garnet
- [4] Haystack