Vivid: A Framework for Creating Visual Programming Languages

Simon Gauvin
(gauvins@cs.dal.ca)
Tel: 902 489-7003

Trevor Smedley
(smedley@cs.dal.ca)
Tel: 902 449-6944

Topic Area: Frameworks, Software Architectures

Faculty of Computer Science
Dalhousie University
6050 University Avenue
Halifax, Nova Scotia
Canada
B3H 1W5
Vivid: A Framework for Creating Visual Programming Languages

Simon Gauvin
(gauvin@cs.dal.ca)

Trevor Smedley
(smedley@cs.dal.ca)

Faculty of Computer Science, Dalhousie University, Halifax, Canada

Abstract
We present an approach for the creation of visual languages through the use of a software application framework. Vivid is a framework that provides a platform for the creation of, and experimentation with, visual programming languages using a modular approach based on abstracting elements of the language as components. By creating self contained components, language designers can combine language elements within the framework to quickly experiment with new visual language designs.

1. Introduction
Software development is a complex and time consuming activity. The development community has known for a long time that the reuse of existing code is a good way to reduce development time. One approach that has demonstrated success in reuse is Frameworks [7]. A framework is a reusable design of all, or part, of a software system. Frameworks provide a partial solution to a problem that allows a developer to complete the solution themselves. Frameworks therefore provide some code that addresses the problem directly, thus reducing the overall amount of code required. The resultant effect is that developers can focus on the code that is specific to their problem.

The use of Frameworks has become popular for many areas of software development. The most famous use of frameworks comes primarily from the application development domain, where frameworks were designed to provide developers with ready-made skeletons for applications [10, 15]. Operating systems, such as NeXT [12] and MacOSX [9], have made extensive use of frameworks to provide developers access to every layer of the system. The MacOSX application framework, Cocoa [1], even provides visual tool, Interface Builder, for automating the creation of code that extends the framework.

Frameworks are not limited in scope to just the domain of application programming. Since frameworks are a partial solution to a problem they can be applied to many domains. The use of frameworks has also been applied to other domains such as mathematics, physics, 3D graphics, networking, etc.

The development of visual programming languages (VPL) is a unique domain that incorporates many sub-domains of computer science. A VPL involves the use of code for applications, graphics, user interface design, compilers or interpreters, and graphs. This collection of requirements increases complexity to the point where we find that general application frameworks are not sufficient. A general application framework is a partial design of an application, not a partial design of a VPL.

One approach to the development of VPLs with frameworks would be to combine individual frameworks, each of which provides some of the requirements of a VPL, when combined could provide the full set of requirements. However, integrating frameworks is often very complex due to the nature of framework design. This is because framework designers make design decisions that isolate the frameworks inner workings from the developer in order to simplify development. This isolation often results at the expense of inter-framework integration. This complexity produces unwieldy interconnectivity resulting in brittle software.

One of the ongoing problems in visual language research is that very few of the languages developed reach the stage where they can be shared, and experimented with, by researchers other than the team involved within a project. This results in added difficulty to co-produce and experiment with VPLs between researchers in an age where collaboration is common. This stems from the fact that most VPLs created in research projects are developed using either non-portable languages or systems. This has a negative impact on research as VPLs cannot be shared among researchers limiting knowledge exchange. As well, the implementation of VPLs in research projects are often too specific to one language design that the code is not reusable for other languages. This means that each new VPL requires an entire re-write, again slowing the progress of research in this area.

Some researchers have tried to address the problem of the tedious work required to create VPLs by creating systems [2, 11] that automate the creation of VPLs. However, these again have problems of their own. They are either developed to produce ‘toy’ languages, providing limited research scalability, or are so specialized that the languages they produce are limited to only small ranges of research.

A possible solution would be to create a framework that deals directly with the domain of VPL design. With this more direct approach researchers would be
able to take advantage of a framework designed to support the unique requirements that VPLs present. This approach is based on the observation that the vast majority of the implementation problems that researchers need to deal with are repeated each time a new language is built. Problems such as graphics, connectivity of elements, drag and drop, clipboard use, marquees, and other "common" VPL techniques are not unique. What is unique is that these requirements are not dealt with by a general application framework.

We propose an approach to the development of VPL that is founded in the use of frameworks. We present the Vivid Framework, a framework-based solution to the problem of VPL development that is designed to be reusable, portable, and scalable. Vivid serves as a platform for further VPL research and a medium for research collaboration.

2. Vivid Framework

The Vivid Framework is a framework-based solution to the problem of building VPLs. Its uniqueness lies in its ability to integrate relevant domains into its design that concern VPL researchers and developers. In this section we will provide an overview of the architecture and its unique use of components and commands.

2.1 Architecture

Vivid provides an architecture that encompasses the use of compilers, editors, language semantics, language models, graphics, and user interaction into a complete framework. The result of using Vivid yields a complete integrated development environment around a VPL.

The initial implementation of Vivid uses the Java programming language [4]. We chose Java as an implementation language because of its portability, its rich set of libraries, and its pervasive appearance in most research institutes around the world.

The architecture of frameworks traditionally comes in either two flavors, white-box or black-box [8]. White box design promotes the use of a framework through inheritance as a mechanism for reuse, and for adding domain specific code. This requires developers to understand the "hot spots" in the framework [8] (locations in the framework that expect to be extended), and its internal structure. The black-box approach is less complex and treats the framework as a set of components. This promotes reuse through the combination of domain specific components.

The design of Vivid embraces both of these approaches. Language designers can extend the classes within the framework to create new language elements via the white-box approach. These elements, implemented as components, can then be reused and combined in the framework to create new languages.

The architecture of Vivid organizes the framework into a set of Java Packages (Figure 1) that are separated by function. Each package provides a partial solution to the area of functionality it deals with. For example, the *vivid.model.* package provides a design for the creation of internal application models that deal with the unique requirements of VPL. The classes within this package provide a partial implementation of a model. The model package provides functionality for construction, modification, notification and interfaces for viewing and serializing the model. Developers can thus extend the hot spots in the package to create language specific models.

![Figure 1: Vivid Package Structure](image)

Like general application frameworks Vivid provides a fully functioning application structure that includes windows, menus and editors. However, the application part of the framework is loosely coupled to any specific user interface design and thus allows full integration of Java's JFC [5] for user interface development. This allows language designers to get as creative as they like, and not restricting them to a certain set of widgets, enabling the visual design of the language to be scalable.

2.2 Component Model

The implementation of the Vivid black-box approach is through the use of components. Components are self contained collections of code that can be connected together with other components to form systems. The Vivid framework makes use of components for two purposes: as extensions for connecting external functionality as plugins [8], and for the creation of visual language elements.

Components in Vivid are implemented as JavaBeans [6], the component model provided by Java. Components make the development of systems modular by allowing components to be loosely coupled. In Vivid this property is used to extend the framework with implementation specific functionality, or plugins. The framework is designed to allow white-box extension implemented as
components for black-box assembly of a visual language. For example, the use of a specific compiler or interpreter is implemented using a component by extending classes in the **vivid.lang.** package and bundling the classes into a JavaBean that is specialized to access a specific compiler. This JavaBean is then placed in a designated path, known to Vivid, and can be dynamically loaded to provide access to a specific compiler that the JavaBean interfaces with. Using this black-box approach components can be used to provide various configurations to unique resources with no recompilation of the VPL in use.

One of our goals was to provide a framework that promoted the collaboration between researchers such that their work and implementations could be shared and reused to investigate new ideas about visual language design. A unique feature of Vivid is the use of components where elements of a visual language, such as the graphical syntax, are implemented as components. Using these sets of components a language designer can create new languages. This is possible because the language implementation is decoupled from framework and can be integrated, or shared, with other languages in the same framework.

Starting with the Vivid framework packages, and using a white-box approach, researchers implement language components using several packages of the framework at once. This horizontal approach to implementation yields a component that possess a visual representation, an internal data model, and the control that enables other components to be manipulated. The design of these language components follows the Model-View-Controller pattern. We call these Visual Language Components (VLC) as they are designed to be the basic building blocks for visual languages in Vivid. Each VLC is loaded dynamically at run-time by Vivid and used to create the elements of a language. This allows the VLCs to be decoupled from the framework that knows nothing of their specific implementation. Each VLC allows three separate relationships to be established between itself and other VLCs. This includes the has-a (containment), is-a (inheritance), and uses-a (delegation) relationships most commonly found in VPLs. Inter-component communication is provided through the JavaBean **Listener** interface that decouples components from each other using a mechanism of dynamic registration and notification. The construction of these relationships is provided by other components that establish themselves as parents of the VLCs.

Each of the VLCs are designed for recursive composition. For example: an editor is itself a component that is implemented as a JavaBean that would then be the parent of other child VLCs that act as the visual language elements themselves. Changes to rules, such as how the editor component treats new VLCs, are easily modified through the framework. The generality of Vivids implementations allows any VLC to exist and be combined with any other VLC providing experimentation with no direct changes to a VPLs code. This design provides a rich environment for researchers to share code, via components, that can be easily transported across networks and run on multiple systems given the portability of Java.

### 3. Example Language: JGraph

There is no better way of learning if a framework is well designed then using it directly. Part of our research involves creating visual programming languages, and therefore, we were the first customers of the framework itself. Our first experiment was to implement one of our own visual programming language, JGraph [16].

JGraph is a VPL that is a derivative of Prograph [13], but adapted to follow the semantics of Java. The idea is that JGraph and Java should be easily translatable. Using JGraph allows programmers to take advantage of the simplicity that visual programming provides, yet also take advantage of the portability and support that Java presently enjoys.

The first implementation of JGraph was written in Java using the general application frameworks provided by the Java language. It took a year, using three separate students on work terms, to implement the rudimentary application shell, drawing, interaction, serialization, and basic translation. The code base encompassed 93 files, that totalled approximately 100,000 lines. Upon analysis we saw that 60% of the code dealt with the user interface, 20% was for the internal model, 10% for compilation and translation, and the last 10% was utility code. This means that only 20% of the code actually dealt with the problem of building JGraph itself. Had the other 80% of the code been provided, then we estimate that JGraph would have been completed in approximately 3 months.

One problem that we observed, after analyzing the Java code, was that very little of the code was designed specifically for the JGraph language. Most of the code was written as scaffolding (windows, menus, interaction, serialization, etc.) or was general in terms of solving computations and visual element layout. However, a result of the code design, which depended specifically upon this applications internal structure, very little code was actually reusable for implementing other VPLs. This reinforces our point that designing code for reuse must be a requirement from the start of a project, and cannot be something
that is thought of midway, or attempted with code that was not designed with this requirement at the onset. As well, this result shows that using a general purpose framework, such as the JFC, does not provide significant gains for VPL development.

Using Vivid we can provide dramatic improvements over the design and implementation of JGraph when compared to its present implementation. By providing pre-built functionality for interactivity, graphics, windowing, editing, and visual element design, we can reduce the complexity of the code as well as the time needed to complete it. For example, to see how we can implement a JGraph method editor, and accompanying language elements, see the illustration in Figure 2.

![Figure 2. JGraph Language Components](image)

Using Vivid requires that each language element be a component, a VLC, and that the editor itself be a component, a VLC. A VLC in Vivid requires extending three packages: the language, model, and view. Each extension of Vivid requires a VPL specific implementation that encompasses only the code that defines that part of the language. This means only the top 20% of the code is written, the code that is specific to a particular VPL.

In contrast to this, when we analyzed the code that the first implementation of JGraph produced, very little of it was specific to the method editor and language elements themselves. Most of the code dealt with drawing, interaction handling, window and menu construction, and layout management. As well, each element in the language is produced from a combination of classes that are not necessarily separate, so the coupling is quite tight.

4. Concluding Remarks

The creation of visual programming languages is complex, yet through the use of frameworks and components we have shown a method for developing VPL that is potentially faster and easier. The Vivid framework encapsulates the domain specific requirements and knowledge of visual language design. It provides a reusable platform that is portable and scaleable for on-going research development, yet allows researchers to experiment with different VPL ideas using a loosely coupled components for language construction. With the popularity of Java, and the JavaBean component model, Vivid will be available to the large base of developers. Using Java and its external native interface (JNI) also allows Vivid to integrate with legacy VPL systems.

Our future work will be to complete and publish the Vivid Framework as open source and make it available at the VIVID web site (http://vivid.cs.dal.ca).

5. References