Plugging In and Into Code Bubbles: The Code Bubbles Architecture

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI. 02912
spr@cs.brown.edu

Jared N. Bott, Joseph J. LaViola Jr.
Department of EECS
University of Central Florida
Orlando, FL 32816
{jbott,jjl}@eecs.ucf.edu

Abstract

Code Bubbles is an attempt to redefine the user interface for an integrated programming environment. As it represents a whole new user interface, implementing it as a plug-in is inherently difficult. We get around this difficulty by combining two different plug-in architectures, a standard one based on registrations and callbacks, and a message-based one that puts the plug-in at arm's length and defines a narrower two-way interface. This paper describes both how we have implemented Code Bubbles as a plug-in to Eclipse and how Code Bubbles itself is implemented as a set of plug-ins representing the different aspects of the environment, using both traditional and message-based plug-in architectures as appropriate. It also shows how the resultant architecture is flexible enough to support collaboration, different back ends, and a cloud-based environment.

1. The Architecture of Programming Environments

For over 25 years, integrated programming environments have been built as a series of plug-ins [38]. Today's commercial and standard environments, Visual Studio, Eclipse, and NetBeans, all accommodate new tools as plug-ins. Thus, when we started to develop a new environment framework we naturally wanted to develop it as a plug-in.

Plug-ins have been widely used in programming environments and are common in software including web browsers, document systems, development environments, computer algebra systems, web servers, and more [9,15,24,30,36,42]. Developing programming tools as plug-ins has many advantages. The principal one is that the tool can make use of the overall framework and previous tools. Framework support can include user interfaces, error interfaces, editor support, and access to precomputed program analysis in the form of annotated abstract syntax trees, symbol tables, etc. This approach tends to ensure that the tools conform to the standards of the rest of the environment, for example, by having a common user interface, common means for accessing the tool, and common means for displaying errors and results. It can greatly simplify the plug-ins since each tool does not have to do its own analysis to keep its internal data structures up to date as the programmer edits the code.

At the same time, there are potential disadvantages to developing new tools as plug-ins. A tool designed as a plug-in might be limited to a particular programming environment; it is generally difficult to use an Eclipse Java plug-in with NetBeans for Java for example. Plug-in tools need to conform to the user interface standards of the environment; for example, tools in Eclipse gener-

ally need to use SWT as their graphics package. Unusual tools might find the environment was not designed to accommodate them as plug-ins and their implementations can thus be very difficult. Tool implementations may interfere with other features of the environment, for example a tool that does run time monitoring and instrumentation may interfere with debugging. Tools might have resource demands that exceed those of the underlying environment, for example requiring considerable memory or computation. Tools also have to be debugged, and crashing one's programming environment can leave the source and target system in an unstable state.

There are multiple ways that plug-ins can be implemented in an environment. Most of today's environments use a modified "toaster" model [10]. Here the plug-in is designed to fit against a standardized backbone which includes the program interface for initializing and running the tool, access to standard data structures such as the project, source files, syntax trees, semantic information, and ways of adding buttons, panels, and other user interface components to the standard environmental interface. This is augmented with an event register-callback mechanism that the plug-in can use to act based on events in the environment, for example, when an error is found, when the program reaches a breakpoint, when a file is saved, or when the user clicks on a button.

Eclipse is a good example of such an environment [25]. Eclipse uses the OSGI dynamic component model [1] to organize its plug-ins. This model provides extension points that plug-ins can implement or provide. Extension points act as callbacks and call-ins to the plug-in. Plug-ins can also access other plug-ins directly. Each plug-in consists of a file that contains the executable code, a description of its dependencies (so the direct calls can be bound correctly), the extension points that it uses, and any new extension points that it provides. The OSGI implementation handles automatic loading, binding, and access to the extension points. Extension points are provided for most user interface interactions (e.g. buttons, windows, dialogs, panels). Extension points are also provided for features such as quick fix and refactorings. The base Eclipse implementation itself is minimal, and most of its features, including items such as basic Java support, are implemented using plug-ins.

Eclipse generally has been a very successful example of a plug-in architecture. The basic framework has served as an IDE for a variety of languages including Java, Python, and C/C++ and for different forms of web development. The framework is also widely used in research projects, providing a way of experimenting with new tools, many of which, like Mylyn [18], eventually become part of the environment. Other items, such as IBM's Jazz (Rational Team Concert), are plug-ins to some extent, but require a significant amount of external code and effort. Integrating with tools such as SVN requires adding multiple plug-ins.

While the toaster-model of plug-ins is the most common, it is not the only approach to developing an extensible programming environment. We used an alternative approach using a message bus in the FIELD environment [31]. A message server ran as a separate process and all tool interaction was done via messages sent on sockets to this server. Each tool was run in its own process with a small wrapper that interfaced with the message server rather than running all the tools as plug-ins in a common process. Tools registered patterns describing the messages they were interested in with the message server. Tools then sent messages to the message server. These were either messages designed as commands for other tools, or messages that might be informative to other tools. The message server in turn would forward the messages to all clients that had registered matching patterns. Messages could have associated replies, which would then be returned to the original

sender through the message server. Messages in this context were used both for notifying all tools of interesting events such as a breakpoint being reached, and to create flexible, standardized interfaces to the various tools. For example, the editor could request or send information to the debugger without knowing anything about which debugger or its particular implementation.

Message passing has become a common method of interprocess communication and is used in a wide variety of areas including parallel computing, distributed computing, object oriented programming, and event notification systems. For parallel and distributed computing systems where processing units do not have access to shared memory, messaging passing is commonly used to facilitate inter-process communication [21]. MPI is a widely used message passing communications language for parallel computing [14]. Message passing is important for microkernel architectures, which are composed of many small programs that rely on high speed inter-process communication [20]. Message passing is also used to allow components in plug-in architectures to communicate [40,43]. Some programming languages have special support for message passing. APRIL is a programming language for building distributed applications that uses message passing for inter-process communication [22]. In APRIL, processes subscribe to messages by listing a set of patterns they are willing to accept, similar to the techniques used in FIELD.

While a message-based interface provides less information and facilities to the individual tools than a toaster-type interface, the looser coupling has the advantage of being more flexible and allowing more tool independence.

2. Code Bubbles

Code Bubbles [2,3] is an attempt to redesign the user interface to programming by making the programming environment conform to the programmer's working model by displaying and manipulating complete *working sets*, collections of task-relevant fragments including code, documentation, test cases, notes, bug reports, and other aspects of programming [19,23]. The fragments in a working set may be contained in multiple files, classes, or other modules, so quick and easy viewing of a working set is complicated in traditional IDEs. Code Bubbles presents fragments in fully manipulatable interface elements in order to provide an intuitive arrangement of working sets.

One goal in using working sets and fragments is to abstract away the concept of files. Other research into working sets and programming environments has been performed. Early explorations of similar concepts are found in Desert [32], IBM's Visual Age, and Sheets [39]. Desert is a suite of programming tools that includes support for fragment manipulation. Sheets divides code into fragments; Sheets groups fragments into collections called sheets. In contrast to Code Bubbles, fragments may be contained in multiple sheets. Additionally, the Sheets Hypercode Editor did not explore spatial arrangement of fragments and was limited to code-based fragments (although they discussed non-code fragments). More recently, Microsoft's Code Canvas is a zoomable user interface that operates as a front-end to an IDE [7,8]. Like Code Bubbles, it provides a workspace for arranging and viewing code documents. Code Canvas allows the user to zoom in and out of their workspace, providing different capabilities at different zoom levels. For instance, a user zooms in to obtain editing capabilities. In contrast to Code Bubbles, Code Canvas provides an initial spatial arrangement of fragments and allows the user to rearrange the fragments. LightTable is a programming environment that also uses fragments and spatial arrange-

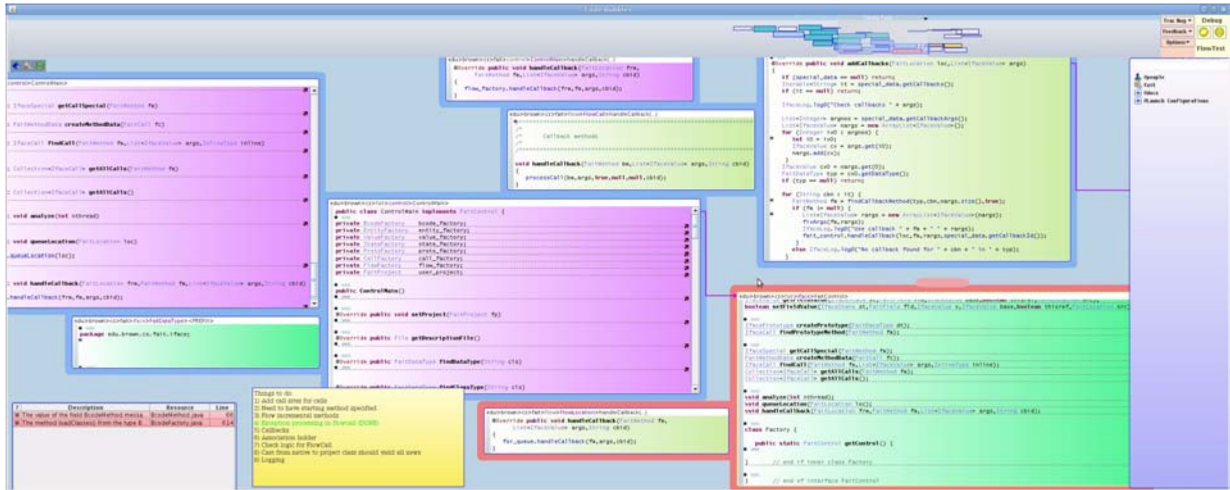

FIGURE 1. A view of the Code Bubbles Environment. This view show multiple bubbles. The bubble in the lower left is a fixed bubbles showing the current errors and warning. The bubble next to it is a not bubble with notes the programmer has written about the displayed code. The remaining bubbles are source bubbles showing either individual methods or small classes. The backgrounds are color coded by their package. The bubble on the right is the package explorer which is a searchable view of all functions and classes as well as documentation and launch configurations. The gray area near the top is an overview of all the bubbles that the programmer has opened in the session.

ment of fragments [13]. CodePad is a multitouch display designed to allow developers to manipulate fragments [28]. CodePad uses gesture recognition to provide gesture-based interaction with an IDE and manipulation of fragments. Code Bubbles allows developers to manipulate fragments, but uses a special front end to an existing IDE and does not provide multitouch interaction. Our goals in developing Code Bubbles also differ with CodePad. In comparison to Code Bubbles, CodePad is focused on providing touch-based interactions with an existing IDE. Some programming languages have also abstracted away the concept of files, such as Smalltalk [12], Squeak [16], and Self [41].

Numerous IDEs have been designed to visualize code. Some focus on learning different parts of programming, such as data structures or object-oriented programming [17,27,29,35,37]. Others, such as Code Thumbnails [6] and JASPER [5] are designed to help with code navigation, an important and complicated task [23,26]. The spatial arrangement of bubbles in Code Bubbles is designed to help with code navigation through spatial memory.

Code Bubbles itself was designed to provide a new user experience, replacing the whole user interface of a traditional programming environment with a bubble-oriented interface that allows more relevant information from the working set to be displayed and provides for a graphical organization of that information. It provides the programmer with access to logical fragments of the source rather than to complete files. It was designed to completely replace the current interfaces while providing the full capabilities of a modern development environment. A view of the environment can be seen in Figure 1.

These characteristics made Code Bubbles a poor candidate for a plug-in tool. Plug-in frameworks are designed so that plug-ins use the user interface of the existing environment, which Code Bubbles does not do. These frameworks assume that the basic environment has a visible user

interface, which again is not the case for Code Bubbles. Existing plug-ins in a framework often assume that they can use the existing user interface, which make them difficult to use with Code Bubbles and limit the ways in which Code Bubbles can plug in. However, in writing Code Bubbles we did not want to reinvent the wheel. We were interested in exploring a new user interface for programming, not a new programming environment. We wanted to build on top of an existing framework and not have to reimplement project management, program analysis, compilation, etc. This meant we really wanted to implement Code Bubbles as a tool plug-in to an environment.

In designing and developing Code Bubbles, we had several goals in mind beyond that of creating a new user experience and user interface for programming. In particular we wanted to:

- Support multiple languages. While demonstrating that the environment worked for Java was useful, we were also interested in how this type of a user interface for programming worked for non-object oriented languages, for dynamic languages, and for systems involving mixed languages, for example web applications.
- Support multiple environments. While Eclipse is widely used in academic circles, other environments are more widespread in industry. Switching projects, especially complex projects, from one environment to another can be quite difficult. We wanted to enable our user interface to be used with multiple environments without having to rewrite a significant amount of code.
- Support collaborative development. The Code Bubbles interface is essentially a single programmer's view into a system. Since most software is developed in teams, we were interested in what tools, techniques, and frameworks would be needed to support collaborative development. In particular, we wanted to look at supporting agile development through pair programming, supporting team-based development when using a version management system, and providing support for team communication during development.
- Support cloud-based development. We wanted to explore how to run our programming environment in the cloud, taking advantage of cloud-based facilities for compiling, testing, file storage, etc., while working on a computer that might be logically or physically separate. For example, developers might want to maintain a single source environment, but work on that from both their office and their home.
- Support CPU-intensive programming tools. We also wanted to explore a variety of tools that could support programming but might be too expensive to include in the actual programming environment. This would include continuous testing, in-depth static analysis to find potential problems, and the use of machine learning techniques to either help the programmer code or to identify potential problems.
- Support student programming. We wanted to explore how the environment can be used not just for professional, experienced programmers, but also as a means for teaching programming.

Our assessment of these tasks was that they are handled to some extent by current environments, but not particularly well. For example, Eclipse supports multiple languages, but Java still has significantly better support. There are various frameworks for supporting collaboration on top of Eclipse, for example IBM's Jazz (Rational's TeamConcert), but these requires considerable external setup and often seems external to the underlying environment. The effort at putting Eclipse into the cloud and similar efforts such as Mozilla's Skywriter and Cloud9 are hampered by the use of a web-based front end and limited screen space.

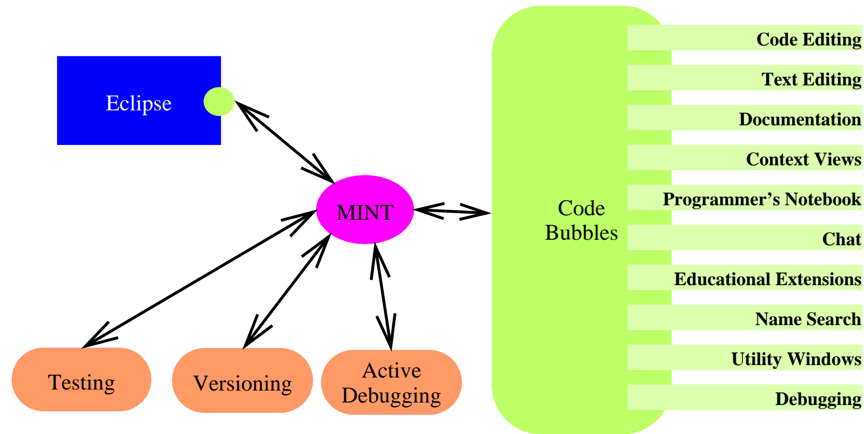

FIGURE 2. Overview of the Code Bubbles Architecture

Code Bubbles itself is a programming environment that can support new programming tools. This led us to design the system to support its own plug-ins and to implement much of the environment in terms of these plug-ins.

3. The Code Bubbles Architecture

In designing and implementing Code Bubbles, we wanted to take advantage of as much of the underlying, non-user interface framework of the underlying environment as possible, while not committing to or forcing ourselves to use the existing user interfaces. We wanted to develop Code Bubbles to work with multiple underlying environments. We also wanted to have the additional flexibility to support collaborative and cloud-based versions of Code Bubbles.

The solution we developed was to first design a message-based interface between the underlying programming environment and Code Bubbles and then to implement an actual plug-in for the environment that interacted with Code Bubbles through messages. The messaging system, MINT, was modeled on the one used in FIELD, updated to use XML messages and patterns and simplified message handling. MINT is a standalone package and has been used in other systems as well. The message-based interface was designed to be independent of both the target-language and the environment. The plug-in itself was designed to use messages both to accept commands from the front end and to relay any notifications available from the underlying environment to Code Bubbles.

An overview of the architecture can be seen in Figure 2. Code Bubbles is a separate process. It includes a variety of plug-ins of its own, with much of the actual user interface work outside of the actual manipulation of bubbles being done in the plug-ins. These plug-ins use the interface features of the environment, for example light-weight windows displayed as bubbles. Most of the low-level work of a programming environment, for example compilation, error checking, search, and managing programs being debugged, is handled by the underlying programming environment, Eclipse in this case. Code Bubbles includes a small Eclipse plug-in as an interface to this functionality. Code Bubbles talks to this plug-in by going through the MINT message bus. Several Code Bubbles plug-ins involve substantial or outside computation. These are implemented as separate processes or outside plug-ins that use the message bus both to interact with

Command	Purpose
OPENPROJECT	Open a project if necessary, list detailed project information (e.g. files, paths, classes)
BUILDPROJECT	Build a project, optionally doing a full build or clean
GETALLNAMES	Return all top-level names (used to set up a search interface in Code Bubbles)
FINDDEFINITIONS	A general search interface for finding definitions (returns locations)
SEARCH	A general text search interface (returns locations)
START	Start debugging a particular debugging configuration
DEBUGACTION	Take a debugger action (continue, step into, step over, terminate, pause, ...)
GETSTACKFRAMES	Get the stack frames for a given stopped thread
EVALUATE	Evaluate an expression; special commands are available to evaluate variables
EDIT	Edit a particular open file (edits include insertions and deletions)
GETCOMPLETIONS	Get suggested text completions
ELIDESET	Define the relevant portion of the source for which elision information is needed
RENAME	Handle variable renaming

FIGURE 3. Some of the Basic Message-Based Interface used by Code Bubbles

Code Bubbles and to obtain information from and utilize the back end. Here three such plug-ins are depicted, one for handling continuous and on-demand testing, one for handling collaborative version management, and one for active debugging.

This architecture is designed to be flexible to meet the various goals for the project. It is possible to replace Eclipse and the Eclipse plug-in with another back end with its own plug-in that handles the same set of messages. This is what we have done in our preliminary implementation of Pybles, Code Bubbles for Python which is described in more detail in Section 6. It is possible to run multiple instances of Code Bubbles with the same message bus and same back end, thereby enabling collaborative programming efforts as described in Section 9. It is also possible to run Code Bubbles on one machine and to run the back end, notably Eclipse and the external plug-ins, on a separate machine. With some care, this can be used to run the back end in a computation cloud and run the front end on the user's machine as discussed in Section 11. Finally, by varying the Code Bubbles plug-ins, we can specialize the environment, for example producing a version of the system suitable for teaching as described in Section 10.

4. The Code Bubbles Message Interface

The message-based interface used by Code Bubbles is designed to facilitate interaction between the Code Bubbles front end and the underlying environment. It is meant to support multiple front ends using the same back end simultaneously as well as users who want to work with both the back end (Eclipse) and Code Bubbles simultaneously.

A sampling of message-based commands is shown in Figure 3. All messages and replies are coded in XML. The commands that return locations return a list of location descriptions that include the file, start and end positions, and, where relevant, the underlying symbol. Commands that return other information, such as a project description, use an XML format that attempts to be independent of the particular information that Eclipse provides.

In addition to commands, the message-based interface includes notifications from the environment. A sampling of the notifications are shown in Figure 4. These are of two types. The first are reflections of events that are occurring within the environment, for example edits done in the

Notification	Description
ELISION	Return an approximation to an abstract syntax tree for elision after an edit occurs or on demand
EDITERROR	Return the error messages for a file after the file has been edited
EDIT	Pass along any file edits done within the environment or by other Code Bubble instances
RUNEVENT	Note changes to a running program (e.g. breakpoint reached, started, terminated)
NAMES	Used to pass back a list of names from GETALLNAMES asynchronously
CONSOLE	Pass along output to either standard out or standard error
EVALUATION	Pass the result of an asynchronous evaluation request
RESOURCE	Note changes to files in the project

FIGURE 4. Some of the Message-Based Notifications used by Code Bubbles

Eclipse editor, files being saved, error messages being created or removed, breakpoints changed, or the state of a program being debugged changed. The other type of notifications is used to support asynchronous commands. Some commands, for example *EVALUATE*, can take a significant amount of time. Rather than pausing the interface here, the commands trigger a background computation where the result is sent back using notifications.

The *ELISION* notification returns a simplified abstract syntax tree. Abstract syntax trees are generally language-dependent. Code Bubbles, however, only needs them internally to determine the hierarchical structure of the code to support code elision for display. As such, the information passed back is a simplified language-independent abstract syntax tree that only includes the high-level nodes that might be elided, an estimation of the priority for elision that is computed by the plug in for these nodes, and additional information about identifiers so that the various types of identifiers can be formatted differently.

The plug-in support for editing is also oriented to providing adequate performance while minimizing messaging. Edits done in the front end are passed to the back end using the *EDIT* command. This causes the back end to make the associated changes in the file. Such changes can create or remove error messages and change the structure of the underlying abstract syntax tree and hence require an *ELISION* notification. Each edit command includes a serial number. The notifications related to the *EDIT* command are only sent if a given time interval (250ms default) has passed since the command was received and no other *EDIT* command has been received for the same file.

The back end also serves as a clearinghouse to facilitate collaborative development using Code Bubbles. Multiple instances of Code Bubbles can start up and communicate with the same back end using the message bus. Each of these can open and potentially edit files from the same project. In addition, the back end, in this case Eclipse, can also be actively run and used for editing as well. The Code Bubbles Eclipse plug-in tracks all active open instances of a file. When an edit command comes in from any of them, including the back end, the edit is made in the back end file and is sent to all the connected front ends that have the corresponding file open.

The notification messages provide additional abstractions to simplify the implementation of Code Bubbles. It first provides a notion of problems that can be defined or removed on a per-file basis. Ideally the back end would track individual error messages and would indicate when they were created or removed. However, with multiple active editors, this becomes impractical since there are multiple buffers (one per active version of Code Bubbles plus one for Eclipse) which might

have slightly different errors along with many shared ones. Instead, messages are identified by file along with their type and file position.

Another abstraction is provided for debugging. Here the back end provides information about debugging runs, process groups, processes, threads, and stack frames. These parallel the abstractions provided by Eclipse and have proven suitable for other languages such as Python.

5. Plugging Code Bubbles Into Eclipse

The Code Bubbles Eclipse plug-in is designed to work in a variety of Eclipse configurations. It will work with standard Eclipse; it will work as a rich client program (i.e. running Eclipse without a front end [11]). We found, however, that neither of these was sufficient for our purposes. We felt that Code Bubbles was a complete front end and that Eclipse shouldn't be needed or visible; however, several Eclipse features such as auto completion, quick fix, and refactoring require the presence of a front end, even if it is not visible or accessible. Therefore, we created another mode whereby the plug-in runs with full Eclipse, but automatically makes all Eclipse windows invisible. This last mode provided us with an Eclipse window context that is needed for some operations and let us pop up Eclipse dialog boxes when necessary (e.g. when a program cannot continue execution after a code fix).

The plug-in itself connects to the message server and registers patterns that match its command syntax. When it receives such a message, it will process it and return the appropriate response. It also registers with Eclipse for callbacks related to run events, launch events, breakpoint events, and resource changes. When any of these events occur, the plug-in interprets the callback to see if it is relevant and generates an appropriate message-based notification if so. A common framework is used to create appropriate XML-based responses for both command replies and notifications in the standard format defined for Code Bubbles messaging.

Most of the command processing in the plug-in is straightforward. The commands are handled using the corresponding Eclipse routines. Unique handles are generated for objects such as breakpoints, launches, and frames, either using internal handles or using object hash codes. Files are identified by their full path names. Each tool talking to the back end is responsible for defining its own unique identifier which is included as part of the command message.

The current implementation is still problematic in some respects. In particular, it has difficulty accessing some of the more sophisticated Eclipse features as well as optional Eclipse plug-ins. We currently are able to access most auto completion information from Eclipse, but have not successfully obtained information from non-standard plug-ins. Similarly, we can handle basic quick fix suggestions for error messages, but have problems with more sophisticated or complex ones that might require user input. Refactorings are even more problematic. We have not been able to successfully integrate with the Eclipse refactoring framework. Instead, we provide a limited set of common refactorings such as rename through lower-level Eclipse calls. This is reflected, for example, in the *RENAME* command rather than a more general *REFACTOR* command.

6. Plugging Code Bubbles Into Python

In order to demonstrate that Code Bubbles can be used for a variety of programming languages and to facilitate its use in today's classrooms, we have been working on developing a Python

implementation called Pybles. We wanted to have an environment that would be easy to install and use for introductory programming. As such, we wanted to have the system be completely self-contained (i.e. we did not want to require that users first install Eclipse or any other environment and then install Code Bubbles).

To achieve this, we developed a Python back end that would be included in the Code Bubbles distribution and that would just work as part of Code Bubbles. The back end was derived from the Eclipse Python plug-in *pydev* by removing all Eclipse dependencies (except for access to the independent Eclipse text processing framework) and simplifying the code as much as possible. In addition, we added a main program that set up a connection to the message server and handled incoming command requests.

The command language handled by the Python back end is the same as that handled by the Eclipse plug-in. This let us use most of the existing Code Bubbles code without modification. The modifications that we have had to make so far include adding Python-specific editor commands such as unindent; different token parsing methods for text highlighting and indentation; a different indentation module; the addition of new symbol types; the addition of new composite code fragments such as *Module*, *Initializations*, and *Definitions*; removing the class hierarchy viewer and the associated semantic analysis package; new code for creating a Python project; and additional creation types for Python modules, classes, and functions.

The message interface is relatively straightforward. The back end builds and caches abstract syntax trees for all the files in the package. These are used to achieve efficient search routines, as a basis for elision output, and for producing appropriate error messages. The back end maintains a each open file and manages locking and notifications when multiple front ends have that file open. In addition, the back end sends notification messages when files are saved or when, upon a reload, new files are found or old ones become obsolete. Background tasks are used to handle incremental compilation and elision output as appropriate.

One of the problems we have been facing with Pybles is exactly what users expect in a programming environment for a dynamic, interactive language. Code Bubbles is currently set up so that program creation and exploration is separate from program debugging. This lets the environment save previous debugging sessions and lets the programmer run different programs or the same program with different arguments in a logical and confined environment. We are setting up the initial version of Pybles with the same framework. However, we are also exploring whether a combined approach where there is a single task that should be executed and that execution task is combined with the main program creation and exploration windows. We hope to report on which is more appropriate based on our experiences in a later paper.

7. Plugging Into Code Bubbles

Code Bubbles itself is designed to be extensible and to support plug-ins of various sorts. In doing so, it has had to provide the necessary hooks and callbacks to enable the easy addition of plug-ins and to maintain the independence of the plug-ins that are used.

There are four core components to Code Bubbles. The first handles low-level operations including properties, file and library access, bug reporting, logging, providing setup, automatic update, and a thread pool for background tasks.

The second provides a procedural interface to the messaging facilities as well as hooks to receive callbacks based on messages received from the underlying environment. This component defines internal structures representing debug-time objects (processes, breakpoints, frames, threads), as well as error messages, code completions, quick fixes, and locations. Locations can either define specific locations in a source file or can be associated with a symbol, either as a reference, a definition, or both.

The third basic component consists of the main program and the facilities for finding, starting and setting up the other plug-ins. Plug-ins are currently defined in a resource file that specifies their main class and optionally, a jar file to load.

The final basic component manages the display. It provides an abstraction of a bubble as a window to be displayed and allows bubbles to be defined as an arbitrary AWT/Swing component. It provides the basic user interface for manipulating bubbles. This includes the overview bar, facilities for defining, saving and restoring working sets, bubble group management, links between bubbles, cursor management, and multiple channels (used in debugging). It provides callbacks when bubbles are updated or removed, when the currently focused bubble changes, when working sets are added or removed, and for when the user requests a "save all". It also provides all the hooks needed to let other plug-ins add to the user interface.

There are two basic ways that plug-ins interact with the user interface. The first is by providing instances of bubbles. A bubble defined by a component needs to provide its contents in terms of a graphic component. It can also define methods for creating an XML representation of the bubble that can later be used to restore the bubble either when the environment restarts or when the bubble is part of a saved working set. Plug-in defined bubbles can also implement routines that are called when the bubble is removed and when the user right clicks on the bubble.

The second approach to interaction involves adding buttons or components to the various menus. Code Bubbles provides a variety of alternatives for buttons including the background right-click context menu, a tool bar that can be quickly created or removed, a permanent button menu in the upper right, and the context menus associated with the top bar overview. In addition, plug-ins can create their own panel at the top of the display. The later facility is used, for example, to create a debugging panel.

In addition to the four core components, Code Bubbles includes several components that are essentially plug-ins but are treated as core components. Two of these are assumed to be present and are initialized by default and can be used by other plug-ins directly. The remaining ones are used by these two components and hence also need to be present.

The first of these is the code editor. In addition to offering full editing facilities, it provides other plug-ins with hooks to add line-oriented annotations, to add items to the right-click context menu, to define tool tips based on the context, and to create new editors given only a method or class name.

The second is the fast search facility of Code Bubbles that lets the user quickly identify a method or class of interest, browse its contents, and then create new bubbles from it. The search facility is designed to be extensible. Other plug-ins can add new dictionaries that will either appear associated with a class or method, or that can appear under a separate heading (e.g. chat buddies). The

Component	Type	Description
Board	Core	Common low-level operations
Bump	Core	Messaging interface
Buda	Core	Basic bubbles user interface
Bema	Core	Main program, plug-in management
Bale	Extended Core	Code editing
Bass	Extended Core	Interactive search box
Bdoc	Extended Core	Documentation viewing
Burp	Extended Core	History (UNDO) management
Buss	Extended Core	Bubble stack (multiple selection) support
Bcon	Plug-in	Context views
Bddt	Plug-in	Debugging bubbles
Beam	Plug-in	Miscellaneous bubbles (notes, flags, ...)
Bedu	Plug-in	Support for using Code Bubbles in the classroom
Bgta	Plug-in	Chat bubbles
Bnote	Plug-in	Programmer's notebook back end support
Bbook	Plug-in	Programmer's notebook front end support
Bopp	Plug-in	Bubbles for setting options
Bted	Plug-in	Text editor bubbles
Bueno	Plug-in	Support for creating new classes and methods
Burp	Plug-in	History management
Batt	Message Plug-in	Testing
Bvcr	Message Plug-in	Version management
Banal	Message Plug-in	Semantic analysis

FIGURE 5. The principal plug-in components in Code Bubbles

facility offers different top level searches (code, documentation, both, all) and new dictionaries can be associated with any set of these.

The third component is the documentation viewer. While this doesn't provide direct facilities to other components, it is implemented as an extended basic component to speed up initialization. The other extended core components, one for history management and one for managing bubble stacks (which result when a search returns multiple items) are required by the code editor. These are loaded as normal plug-ins, but are required to be present.

The rest of the Code Bubbles environment is implemented as independent plug-in components. The complete set of principal components is shown in Figure 5.

Code Bubbles plug-ins are connected to the system using a resource file. While the basic plug-ins are always loaded into the environment, all other plug-ins are loaded only if they are listed in the resource file which can be customized by the local administrator or by individual users. The resource entry for a plug-in lists the main plug-in class, a jar file if the plug-in is not included in the standard Code Bubbles distribution or in the standard plug-in directory, and a listing of which modes (client, server, or normal — see Section 11.) the plug-in is suitable for. When a plug-in is to be used, the system will use Java's reflection mechanism to initialize it. There are two initialization calls, one that is done immediately and one that is called after the windowing and basic plug-ins have been set up.

We have used the plug-in framework provided by Code Bubbles to experiment with new plug-ins and to create differential environments. It serves as the primary mechanism for extending the environment. When we have a new tool or facility we want to include in the environment, we first develop it as a local plug-in that is used only by the developer. Then we include it in the environment distribution and let others activate it if they desire. Finally, when the new tool has been adequately tested, we include it in the standard version of Code Bubbles by adding a line to the appropriate resource file.

The Code Bubbles plug-in framework is not as sophisticated or powerful as that provided by OGSI and used by Eclipse. While it might be convenient to allow new plug-ins just by having them be dropped into a standard plug-in directory, the inconvenience of requiring all dependencies and linkages between plug-ins be made explicit and the problems with dynamic loading and debugging led us to decide, with our small development team, that the simpler mechanism was more appropriate for now.

8. Plug-ins Using the Message Bus

Three of the components are listed as message plug-ins. These are special in that they use both the internal plug-in interface provided by Code Bubbles and the messaging facilities.

The first of these is used for managing test cases. This plug-in consists of two parts. A separate process that talks to both Code Bubbles and to the back end (Eclipse) using the message interface does most of the work. This process uses message-based commands talking to Eclipse to find the set of JUnit tests in the project and how to run them. It tracks resource messages from Eclipse to determine when tests are out of date and need to be rerun. It offers its own command interface to control when the tests should be run, whether to support continuous testing or only testing-on-demand, and to run individual tests. It also sends updates on test status as messages when the statuses are changed. The other portion of this interface is a standard plug-in that can display a test status bubble. In addition to integrating directly with Code Bubbles, this interface creates its own link to the message server to send and listen to messages from the testing process server.

The second message-based plug-in handles version management. It also runs as a separate process that communicates to the back end and Code Bubbles using the message bus. The process uses message-based commands to determine all the files in the current project, and then tests whether they are under version control, and, if so, gets versioning information. While we plan to have several bubbles associated with this interface, the viewer plug-in for version management still hasn’t been implemented.

The third message-based plug-in handles semantic analysis. This involves scanning the user’s system and reporting all relationships related to the class hierarchy, interface usage, nesting, and use-definitions relationships. This can take a considerable amount of time and space and needs to be done by accessing the compiled files. The information is used, for example, by the context views plug-in (Bcon) that displays a graph of whichever relationships the user selects. Implementing this plug-in using the message bus lets us automatically update information as the user saves and compiles files without affecting the rest of the environment, and ensures that when the environment is run as a server (see Section 11.) the information is computed by the back end.

Another message-based facility provided by the environment involves our enhanced debugging features. When we are debugging a user program, we include a small class as part of the user's code to monitor what is going on. This lets us provide more detailed information on thread states, accurate and low-overhead performance information, fast detection of deadlocks, and additional debugging information for Java programs that use Swing. This code needs to talk to the Code Bubbles environment and provide its debugging information. This is done by a separate server, BandAid, that the monitoring class connects to and that then formats and sends the information from the user process to the message bus and thus to various Code Bubbles tools. In addition, the separate server listens for commands from the message bus (and hence Code Bubbles), and can relay them to the code in the user's program.

9. Plug-ins for Collaboration

Most software written today is developed colloboratively. As such, it makes sense for a software development environment to support collaboration and that is what we wanted to do with Code Bubbles. However, there are many different types of collaboration and collaborative development and the different approaches require different facilities.

At one extreme there is a very loose collaboration where programmers share source through a source control system such as *git* and communicate via e-mail or instant messaging. Code Bubbles supports this through both built-in features and through a combination of plug-ins. The built-in features include the ability to create working set files that can be reloaded by another user and the ability to print and send these files via e-mail. Plug-ins enhance these basic capabilities. The chat plug-in (Bgta) provides chat bubbles using any of the common chat services (Google, AOL, XMPP). It includes the ability to save chat histories and to send working sets and other files as part of the chat. The version management plug-in determines which version management system is used by each of the user's projects, and provides common access to that facility. The source control systems currently supported include *svn*, *git*, and *cvs*. The support for notes in the miscellaneous bubble plug-in allows notes to be attached to source lines and saves both the note and the attachment points so that they can be shared among multiple programmers.

The source control plug-in runs as a separate process, listening to messages from both the back end (Eclipse) and Code Bubbles. Whenever it detects that files have changed, it computes the differences between the current files and the last checked out version. It creates a file describing these differences and sends that file to a central server. When the user opens a new file, the tool requests from the central server, the set of all changes made to that file by other users relative to the version the user is currently working on. Lines that are changed are then marked in the editor by a colored band in the annotation bar and associated tool tips. This provides immediate feedback to the programmer that the method they are looking at has been or is in the process of being changed by other programmers.

While this type of facility is not unique, Code Bubbles makes it work without any setup and makes it work securely. The only configuration that the user needs to accomplish this is to add a private UID file to the top level of the project. This is then used to create a key that is in turn used to encode all the files being sent. This assures that access to the server does not provide access to the original code and lets any server be used securely. User names, project names, etc. are all sim-

ilarly encoded. This is in contrast to systems such as IBM's Rational Team Server or Crystal [4] hat require extensive setup and initialization.

At the other end of the collaboration spectrum, there is pair programming as practiced in agile development where two programmers work simultaneously on the same computer. We wanted to provide a level of support for this type of development. The Code Bubbles architecture lets multiple Code Bubbles front ends talk over the message bus to a common back end. Moreover, the code in the back end supports simultaneous editing and updating from the different front ends. This, combined with chat bubbles allows multiple programmers to work simultaneously on the same code.

This is further augmented with the ability to have shared working sets. A working set here is a designated region of the overall display area that is colored and labeled. A user can mark a working set as shared. Any other Code Bubbles interface working on the same message bus, can then display the shared working set. Any new bubbles, bubble movements, etc. done in any instance of the shared working set are sent to and duplicated in all the other instances. This provides programmers with the ability to do pair programming or to support interactive code reviews.

Multiple instances of Code Bubbles running on a common back end can be used for individual development as well as for shared development. Where screen space is at a premium, users will sometimes have multiple machines. We have used this feature, for example, to run Eclipse and Code Bubbles both on our main (Linux) machine, and, simultaneously, on a Windows machine, thereby effectively doubling our screen space.

10. Plug-ins for Education

Programming environments for education should minimize the time it takes to get a student programming. Students are already dealing with unfamiliar, complex concepts; the programming environment should not be another obstacle to overcome. By providing a minimal set of tasks and user interface elements, we can lessen the student burden in learning the programming environment. We also want to make important tasks obvious; while professional programmers generally have more experience with programing and programming environments, students have less experience, so learning an environment is a larger problem for them. Thus, the student version of Code Bubbles, Suds, provides a simpler environment, in order for students to move past learning the interface to learning programming concepts. Through the use of plug-ins, we can allow students to enable additional functionality as they mature as programmers. This allows students to learn an interface once and not have to learn a new one once they have outgrown the functionality initially provided by Suds.

Suds provides an educational environment both by adding additional functionality through appropriate plug-ins and by removing unneeded functionality by customizing the set of used plug-ins in the student version. The Bedu plug-in provides a variety of tools to facilitate classroom-related activities. The first is a TA chat facility, that lets a student submit a request to chat with a TA, queues such requests, and lets the TA respond to them in the order received. It also maintains the anonymity of the TA's user id so that they are not overwhelmed with questions outside of their hours. The second is a facility to set up a new project for an assignment. This lets the professor or TAs create a project template with appropriate documentation, libraries, test files, support code,

and skeleton student code, and then automatically copies and sets up a corresponding project in the student's workspace. The third facility provided by Bedu will take a student assignment project and will create a homework submission from it, combining the relevant files into a bundle that is signed, dated, and placed in the appropriate submission directory. This is done by running an external script that is specific to the particular course and assignment and can be used to integrate with course submission systems. The fourth facility is course-based help that allows documentation and Code Bubbles help files to be augmented or customized for the course. The feature provides access to web-based surveys or questionnaires that can be course and assignment specific and that can be automatically invoked at certain points, for example after assignment submission, or after certain time intervals to better understand the issues and problems facing the students.

Suds lets the environment be customized on a course-by-course basis. When Suds is run, Code Bubbles is passed a course id. Whenever it attempts to open a resource file, it first checks for a resource file that is specific to that course-id. This provides a mechanism both to restrict the set of plug-ins that are appropriate for a given course and assignment, and to change the properties of the environment to better suit the needs of their students. For example, in introductory course might distribute a configuration that provides a very minimal set of plug-ins. As the course progresses, new configurations would be provided to students that enable new functionality. We think this would help students to learn the new functionality when it is made available to them, because each new configuration would allow them to focus on the selected tools and tasks.

11. Plugging into the Cloud

While most software development environments today are designed to work on individual workstations, computing is moving more to a model with a centralized (or distributed) cloud providing flexible processing power and large amounts of storage. Especially for larger systems or for tools that require considerable processing, e.g. running test cases, semantic or security analysis, optimization, it can be desirable to provide support for the environment in the cloud. Moreover, developers today are used to working on a variety of different machines, for example, in separate offices, at home, or while traveling. As part of Code Bubbles, we wanted to provide support for software development where the programming environment would run "in the cloud" and where the user could run the front end on an arbitrary machine. We assume that the front-end machine has sufficient screen space and appropriate tools (e.g. keyboard, pointer) to support programming, but don't want to assume a direct connection to the back-end or direct access to the back-end files including the source and all generated artifacts.

There are several architectures that could be used to achieve an implementation of this type of a cloud-based development environment. The first would be to run the complete environment in the cloud and redirect the display to user's machine. With networks getting faster, this the use of remote displays has become possible. Several Internet-scale games, for example, are implemented in this way. However, the reality today is that networks outside of a company or university environment, especially wireless networks, are not fast enough to make this practical.

A second alternative would be to run the back end, Eclipse, in the cloud and run Code Bubbles itself on the user's local machine, using the message bus to connect the two. The amount of network traffic required for the messages between the two is such that this would be practical

even for relatively slow networks. While this is attractive, it is also problematic. There are several issues here. The first is that the user will want access to files beyond source files. While the Code Bubbles code editor can retrieve file contents from the back end, the text editor does not. Second, several of the tools integrated into Code Bubbles assume they have access to the user files. These include the testing tool which needs to analyze the binaries to find test cases; the versioning tool which needs to analyze sources to find differences from the last version and needs to call the version management system to get file version history information; the semantic analysis tool which needs to analyze the binary files; the extended debugging tool which needs to communicate directly with the application being debugged; the educational tools, which want to submit assignments to local directories; note bubbles which need to store the notes in the project workspace; and the programmer's notebook which needs to access the common database to store and retrieve its data.

Our solution was a variation on the closely-collaborative version of Code Bubbles. We run two instances of Code Bubbles that are connected using the message bus. The first instance runs in the cloud environment along with the back end. It includes the various plug-ins that need access to the project files, either source or binary, and that can make effective use of the resources of the cloud. This includes the plug-ins for testing, version management, semantic analysis, intelligent debugging, and back end support for the programmer's notebook. In addition, it supports message-bus-based commands to get the contents of a file accessible from the cloud. This server version of Code Bubbles does not put up the standard bubble display. Instead, it creates a small dialog box that the user closes to cause the server to exit.

The other instance of Code Bubbles, the client, runs on the user's machine. It includes all the plug-ins that provide actual bubbles as well as the plug-ins that directly support these bubbles. Where it needs to access a plug-in that is actually running in the back end, it uses the message bus. For those plug-ins where the code is run in a separate process, the only change was that the front-end version of the system did not need to start up the support process. For other instances, such as the programmer's notebook, the front end sends requests via the message bus to the back end, so we had to add both a simple message interface that supported the small set of calls.

To further support cloud-based development, we extended the message bus so that it can work over the Internet using a local web proxy to relay messages. All traffic sent over this connection is encrypted for privacy. This lets us run Code Bubbles as a server at Brown, for example, while running Code Bubbles as a client at home without having to connect directly to the Brown network. Our experience with this type of connection is that it is slightly less responsive than running the whole environment at Brown, but it is still very usable.

12. Conclusions

We currently use Code Bubbles as if Eclipse were not present (i.e. as our primary interface to a programming environment). This demonstrates that the message-based plug-in approach that we utilize is effective and practical. Moreover, the implementation depends heavily on the back end, with very little duplication of effort between Eclipse and Code Bubbles. The only significant duplication is the code for handling indentation for which we need very fast turnaround since it is used to reflow or reformat editor lines on the fly.

While the message-based interface has been designed to be independent of the back end, it has only been used thus far to connect Code Bubbles to Eclipse. We are in the process of creating a new back end for Python and attempting to use the same interface (i.e. same set of commands and notifications), for this purpose. The success of this effort will help us understand whether the current interface is an appropriate one.

The plug-in architecture of Code Bubbles itself has shown itself to be fairly resilient. However, most of the extensions have been done by the same core team of programmers, so the real test of whether the facilities are sufficient and correctly designed will be in the future as others attempt to add their own plug-ins to the open source version of the system.

Our experiences demonstrate the weaknesses of attempting to implement a plug-in that tries to do things within the environment that plug-ins were never intended to do. This is especially true in attempting to make use of other Eclipse plug-ins or facilities such as refactoring which seem to be intimately tied to the user interface even when this might not be required.

Using a message-based plug-in architecture has also led to new opportunities and the ability to use the underlying programming environment in new ways [33]. This includes the ability to do shared work, essentially using the same Eclipse back end to support distributed pair-programming, and the ability to run Code Bubbles as a cloud-based environment. The fact that the testing and version management plug-ins are implemented using messaging lets them easily run in the cloud as well.

Code Bubbles is available from *http://www.cs.brown.edu/people/spr/codebubbles* as a free download. This distribution also includes the Python version (Pybles) and the student version (Suds). The system is open source and is available on SourceForge. A shorter version of this paper originally appeared in TOPI 2012 [34].

13. Acknowledgements

This work is supported by the National Science Foundation grant CCF1130822. Additional support has come from Microsoft and Google.

14. References

1. Alexandre de Castro Alves, *OSGI in Depth*, Manning Publications (2011).
2. Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr., "Code bubbles: rethinking the user interface paradigm of integrated development environments," *International Conference on Software Engineering 2010*, pp. 455-464 (2010).
3. Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr., "Code bubbles: a working set-based interface for code understanding and maintenance," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, pp. 2503-2512 (2010).
4. Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin, "Crystal: precise and unobtrusive conflict warnings," *Proceedings 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pp. 444-447 (2011).
5. Michael Coblenz, Andrew Ko, and Brad Myers, "JASPER: an Eclipse plug-in to facilitate software maintenance tasks," *Proceedings ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications Workshop on Eclipse Technology*, pp. 65-69 (2006).

6. Robert DeLine, Mary Czerwinski, and Brian Meyers, Steven M. Drucker, and George G. Robertson, "Code thumbnails: using spatial memory to navigate source code," *IEEE Symposium on Visual Languages and Human-Centric Computing 2006*, pp. 11-18 (2006).

7. Robert DeLine and Kael Rowan, "Code canvas: zooming towards better development environments," *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pp. 207-210 (May 2010).

8. Robert DeLine, Andrew Bragdon, Karl Rowan, Jens Jacobsen, and Steven P. Reiss, "Debugger canvas: industrial experience with the Code Bubbles paradigm," *International Conference on Software Engineering 2012*, (June 2012).

9. R. Diankov and J. Kuffner, "Openrave: A planning architecture for autonomous robotics," *Robotics Institute*, *Pittsburgh*, *PA*, *Tech*. *Rep*. *CMU- RI-TR-08-34*, (2008).

10. ECMA, "Reference Model for Frameworks of Software Engineering Environments," *Technical Report NIST 500-211*, *ECMA TR/55*, (1993).

11. Erich Gamma and Kent Beck, *Contributing to Eclipse*: *Principles*, *Patterns*, *and Plug-ins*, Addison-Wesley (2004).

12. Adele Goldberg and Dave Robson, *Smalltalk-80*: *the Language and Its Implementation*, Addison-Wesley (1983).

13. Chris Granger, "Light Table," *http://www.lighttable.com*, (2012).

14. William Gropp, Ewing Lusk, and Anthony Skjellum, *Using MPI*: *Portable Paralle Programming with the Message-Passing Interface*, MIT Press (1994).

15. A. Helsinger, M. Thome, and T. Wright, "Cougaar: a scalable, distributed multi-agent architecture," *Proceedings IEEE International Conference on Systems*, *Man and Cybernetics*, pp. 1910-1917 (2004).

16. Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay, "Back to the future: the story of Squeak, a practical Smalltalk written in itself," *Proceedings ACM SIGPLAN Conference on Object-Oriented Programming*, *Systems*, *Languages*, *and Applications '97*, pp. 318-326 (1997).

17. Caitlin Kelleher and Randy Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Computing Surveys* Vol. **37**(2) pp. 83-137 (June 2005).

18. M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for IDEs," *Proceedings Aspect Oriented Software Development '05*, pp. 159-168 (2005).

19. Andrew J. Ko, Htet Aung, and Brad A. Myers, "Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks," *Proceedings of the 27th International Conference on Software Engineering*, pp. 126-135 (2005).

20. Jochen Liedtke, "Improving IPC by kernel design," *Proceedings 14th ACM Symposium on Operating Systems Principles*, pp. 175-188 (1993).

21. Oliver A. McBryan, "An overivew of message passing environments," *Parallel Computing* Vol. **20**(4) pp. 417-444 (April 1994).

22. F. G. McCabe and K. L. Clark, "APRIL: Agent Process Interaction Language," *Proceedings of the Workshop on Agent Theories*, *Architectures*, *and Languages on Intelligent Agents*, pp. 324-340 (1995).

23. B. A. Meyers, A. J. Ko, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering* Vol. **32**(12) pp. 971-987 (2006).

24. Mauro Migliari and Vaidy S. Sunderam, "PVM emulation in the harness metacomputing system: a plug-in based approach," *Proceedings 6th European PVM/MPI User's Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 177-124 (1999).

25. Kim Moir, "Eclipse," pp. 89-110 in *The Architecture of Open Source Applications*: *Elegance*, *Evolution*, *and a Few Fearless Hacks*, ed. Greg Wilson,CreativeCommons (2011).

26. G. C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Eclipse IDE?," *IEEE Software* Vol. **23**(4) pp. 76-83 (2006).

27. F. Olivero, M. Lanza, and M. Lungu, "Gaucho: From integrated development environments to direct manipulation environments," *Proceedings of 1st International Workshop on Flexible Modeling Tools*, (2010).

28. Chris Parnin, Carsten Gorg, and Spencer Rugaber, "CodePad: interactive spaces for maintaining concentration in programming environments," *Proceedings 5th International Symposium on Software Visualization*, pp. 15-24 (2010).

29. Anrold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson, “A survey of literature on the teaching of introductory programming,” *Proceedings Working group reports on ITiCSE on Innovation and technology in Computer Science Education*, pp. 204-223 (2007).
30. Alessandro Pedretti, Luigi Villa, and Giulio Vistoli, “VEGA - An open platform to develop chemo-bio-informatics applications, using plug-in architecture and script programming,” *Journal Computer-Aided Molecular Design* Vol. **18**(3) pp. 167-173 (2004).
31. Steven P. Reiss, “Interacting with the FIELD environment,” *Software Practice and Experience* Vol. **20**(S1) pp. 89-115 (June 1990).
32. Steven P. Reiss, “Simplifying data integration: the design of the Desert software development environment,” *Proceedings 18th International Conference on Software Engineering*, pp. 398-407 (March, 1996).
33. Steven P. Reiss, Jared Bott, and Joseph LaViola, “Code Bubbles: A Practical Working-Set Programming Environment,” *International Conference on Software Engineering 2012*, pp. 1411-1414 (June 2012).
34. Steven P. Reiss, “Plugging in and into Code Bubbles,” *Proceedings Workshop on Developing Tools as Plug-ins 2012*, pp. 55-60 (June 2012).
35. Mitchel Resnick, John Maloney, Andres Monrow-Hernandez, and Natalie Rusk, “Scratch: programming for all,” *Communications of the ACM* Vol. **52**(11) pp. 60-67 (Nov. 2009).
36. Steven Ritter, Kenneth R. Koedinger, and An architecture for plug-in tutor agents, *Journal of Artifical Intelligence Education* Vol. **7**(3-4) pp. 315-347 (Jan. 1996).
37. Vineet Sinha, David Karger, and Rob Miller, “Relo: helping users manage context during interactive exploratory visualization of large codebases,” *Proceedings of the 2005 ACM SIGPLAN OOPSLA Workshop on Eclipse Technology eXchange*, pp. 21-25 (2005).
38. Richard Snodgrass and Karen Shannon, “Supporting flexible and efficient tool integration,” *Proceedings International Workshop on Advanced Programming Environments*, pp. 290-313 (June 1985).
39. Robert Stockton and Nick Kramer, “The Sheets hypercode editor,” Carnegie Mellon University (1998).
40. Richard N. Taylor, Nedad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow, “A component and message-based architectural style for GUI software,” *IEEE Transactions on Software Engineering* Vol. **22**(6) pp. 390-406 (June 1996).
41. David Ungar and Randall B. Smith, “Self: the power of simplicity,” *SIGPLAN Notices* Vol. **22**(12) pp. 227-242 (December 1987).
42. Kris De Volder, “JQuery: a generic code browser with a declarative configuration language,” *Proceedings 8th International Conference on Practical Aspects of Declarative Languages*, pp. 88-102 (2006).
43. Anthony I. Wasserman, “Tool integration in software engineering environments,” pp. 137-149 in *Software Engineering Environments*: *Proceedings International Workshop on Environments*, ed. F. Long,Sprinter- Verlay (1990).