University Of Crete
Computer Science Department

CIRCULAR META IDE FOR THE DELTA LANGUAGE:

EXTENSIBILITY LAYER FOR DELTA, DEBUGGER, RUNTIME ADAPTATION AND PROJECT MANAGER

by
THEMISTOKLIS BOURDENAS

Master's Thesis

Heraklion, August 2007

University Of Crete
Computer Science Department

CIRCULAR META IDE FOR THE DELTA LANGUAGE:

EXTENSIBILITY LAYER FOR DELTA, DEBUGGER, RUNTIME ADAPTATION AND PROJECT MANAGER

by
THEMISTOKLIS BOURDENAS

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science

Author: _____
Themistoklis Bourdenas, Department Of Computer Science

Board of enquiry:

Supervisor _____
Constantine Stephanidis, Professor

Supervisor _____
Anthony Savidis, Associate Professor

Member _____
Evangelos Markatos, Professor

Member _____
Mema Roussopoulos, Assistant Professor

Approved by: _____
Panos Trahanias, Professor
Chairman of the Graduate Studies Committee

Heraklion, August 2007

CIRCULAR META IDE FOR THE DELTA LANGUAGE:

EXTENSIBILITY LAYER FOR DELTA, DEBUGGER, RUNTIME ADAPTATION AND PROJECT MANAGER

THEMISTOKLIS BOURDENAS

Master's Thesis

University Of Crete
Computer Science Department

ABSTRACT

Next to programming languages, Integrated Development Environments (IDEs) are considered as the second most decisive factor for effective software development, playing a critical role in the software lifecycle, especially when it comes to medium-to-large scale systems. Recently, IDEs have been treated mostly as collections of task-oriented tools, gathered and managed together under an extensible programming environment, rather than monolithic systems of the all-in-one style.

Sparrow is an IDE for the Delta dynamic object-based language, developed following two key objectives:

- To support extensibility of features, allowing such extensions to be developed using Sparrow, i.e. it is a circular IDE.
- To facilitate open deployment by third parties to build domain-oriented IDEs, i.e. it is a meta-IDE.

In this Thesis, the development of a large part of the Sparrow IDE has been carried out (roughly corresponding to around half of the Sparrow IDE implementation size), addressing the following features and components:

- Software layer, user library (API) and User Interface to support extensibility of the Sparrow IDE in the Delta language.
- Infrastructure and User Interface to support runtime adaptation of the Sparrow IDE to programmer's preferences and requirements.
- Software components for project management support, including the User Interface, extensibility and deployment user libraries (APIs).
- Interactive source-level debugger, including extensibility and deployment user libraries (APIs).

An important aspect of the development process has been the extensive application of the *circular development* style. More specifically, for every component a basic version was initially implemented in C++. Subsequently, once this component was incorporated to the Sparrow IDE, the rest of its functionality was implemented through the Delta language and using the Sparrow IDE itself. Effectively, the Sparrow IDE makes available to its users all of its current as well as future components as a runtime library in the Delta language.

ΚΥΚΛΙΚΟ ΜΕΤΑ ΠΕΡΙΒΑΛΛΟΝ ΑΝΑΠΤΥΞΗΣ ΓΙΑ ΤΗΝ ΓΛΩΣΣΑ DELTA:

ΥΠΟΣΤΗΡΙΞΗ ΕΠΕΚΤΑΣΗΣ ΜΕΣΩ DELTA, ΕΚΣΦΑΛΜΑΤΩΤΗΣ, ΠΡΟΣΑΡΜΟΓΗ ΚΑΤΑ ΤΗ ΧΡΗΣΗ ΚΑΙ ΔΙΑΧΕΙΡΙΣΗ ΠΡΟΓΡΑΜΜΑΤΩΝ

ΘΕΜΙΣΤΟΚΛΗΣ ΜΠΟΥΡΔΕΝΑΣ

Μεταπτυχιακή Εργασία

Πανεπιστήμιο Κρήτης
Τμήμα Επιστήμης Υπολογιστών

ΠΕΡΙΛΗΨΗ

Μετά τις γλώσσες προγραμματισμού, τα ενσωματωμένα περιβάλλοντα ανάπτυξης (IDEs) θεωρούνται ο δεύτερος σημαντικότερος παράγοντας για την ανάπτυξη λογισμικού, ο οποίος παίζει κρίσιμο ρόλο στον κύκλο ζωής του λογισμικού, ιδιαίτερα όσο αναφορά μεσαίας έως μεγάλης κλίμακας συστήματα. Πρόσφατα, τα IDEs αντιμετωπίζονται κυρίως ως συλλογές εργαλείων συγκεκριμένου σκοπού, συσσωρευμένα και οργανωμένα μαζί κάτω από ένα επεκτάσιμο προγραμματιστικό περιβάλλον, έναντι των μονολιθικών συστημάτων όπου συγκεντρώνουν όλα σε ένα εργαλείο.

Το Sparrow είναι ένα IDE για την δυναμική οντοκεντρική γλώσσα Delta, το οποίο έχει αναπτυχθεί ακολουθώντας δύο βασικούς στόχους:

- Να υποστηρίζει επεκτασιμότητα των χαρακτηριστικών του, επιτρέποντας επεκτάσεις να αναπτυχθούν χρησιμοποιώντας το ίδιο το Sparrow, επομένως είναι ένα κυκλικό IDE.
- Να προσφέρει μια ανοιχτή υποδομή για τρίτους ώστε να χτίσουν IDEs τα οποία στοχεύουν σε συγκεκριμένο πεδίο, επομένως είναι ένα meta IDE.

Σε αυτή την εργασία, πραγματοποιήθηκε ένα μεγάλο μέρος της ανάπτυξης του Sparrow IDE (περίπου αντιπροσωπεύει την μισή υλοποίηση σε μέγεθος), το οποίο περικλείει τα παρακάτω χαρακτηριστικά και τμήματα:

- Στρώμα λογισμικού, βιβλιοθήκη χρήστη (API) και διεπαφή χρήστη για υποστήριξη επεκτασιμότητας στο Sparrow IDE από στην γλώσσα Delta.
- Υποδομή και διεπαφή χρήστη για υποστήριξη προσαρμογής κατά τη διάρκεια χρήσης του Sparrow IDE στις προτιμήσεις και απαιτήσεις του προγραμματιστή.
- Τμήματα λογισμικού για υποστήριξη διαχείρισης εργασιών, συμπεριλαμβανομένου διεπαφής χρήστη και βιβλιοθήκες χρήστη (API) για επεκτασιμότητα και χρήση.
- Διαδραστικός εκσφαλματωτής σε επίπεδο πηγαίου κώδικα, συμπεριλαμβανομένου διεπαφής χρήστη και βιβλιοθήκες χρήστη (API) για επεκτασιμότητα και χρήση.

Ένα σημαντικό μέρος της διαδικασίας ανάπτυξης υπήρξε η εκτεταμένη εφαρμογή της φιλοσοφίας της *κυκλικής ανάπτυξης*. Πιο συγκεκριμένα για κάθε λογισμικό τμήμα αρχικά αναπτύχθηκε μία βασική έκδοση σε C++. Ακολούθως, καθώς το τμήμα λογισμικού ενσωματώθηκε στο Sparrow IDE, η υπόλοιπη λειτουργικότητα του αναπτύχθηκε μέσω της γλώσσας Delta χρησιμοποιώντας το ίδιο το Sparrow IDE. Ουσιαστικά, το Sparrow IDE προσφέρει στους χρήστες του όλα τα ήδη υπάρχοντα, αλλά και μελλοντικά τμήματα, ως μια προγραμματιστική βιβλιοθήκη στην γλώσσα Delta.

Ευχαριστίες (Acknowledgements)

Θα ήθελα να ευχαριστήσω τον επόπτη της μεταπτυχιακής μου εργασίας Αντώνιο Σαββίδη για την υποστήριξη της μεταπτυχιακής μου εργασίας και την καθοδήγηση του, καθώς και τον Γιάννη Γεωργαλή με τον οποίο έχω τη χαρά να συνεργάζομαι τα τελευταία τρία χρόνια στο πλαίσιο της εκπόνησης των μεταπτυχιακών μας εργασιών που είχαν ως στόχο την υλοποίηση του Sparrow IDE.

Ευχαριστώ την οικογένειά μου η οποία με προσέφερε την ευκαιρία να επιδιώξω τους στόχους μου χωρίς ποτέ να αμφιβάλλει για εμένα. Επίσης, θέλω να ευχαριστήσω τους φίλους που με συντρόφευσαν αυτά τα χρόνια και βρίσκονταν δίπλα μου.

Table of Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

Integrated Development Environments (IDEs) are powerful instruments effectively increasing the productivity of software developers, collecting together all the required tools into a single consistent development environment. Clearly, the popularity and wide use of programming languages heavily depends on the quantity and quality of the respective supplied tools. Such tools typically include a source code editor, a compiler / interpreter, a debugger and build facilities. It is also common for modern IDEs to support multiple source languages and to incorporate various tools like class browsers, version control systems and GUI editors. Besides programming languages domain-oriented IDEs exist for games and web applications.

The development of an IDE is far from being a trivial task, as it demands the implementation of a broad range of programming tools and their seamless and unobtrusive interoperation. In general such tools look similar across different environments, though customized to target languages and systems. However, redeveloping from scratch IDEs is an unrealistic strategy. Theoretically, the IDE implementer would require a basic extensible platform, a sort of a functional skeleton, on top of which further facilities could be incorporated by means of add-on components. In this Thesis we elaborate on key aspects from the development of an adaptable, circular meta-IDE, emphasizing open deployment, self extensibility and programmer-oriented automatic adaptation.

1.1 Context of work

1.1.1 The Sparrow IDE

We follow a component-oriented approach for the construction of our environment. We provide a platform that enables the construction of an IDE instance by congregation of well defined, self-contained entities (components) on top of a basic skeleton. Sparrow is the major deliverable of this Master's Thesis. It is a full featured Integrated Development Environment for the Delta programming language (1).

Sparrow itself is implemented on the platform as a collection of loosely coupled components that have been implemented to fit in this skeleton. For instance, the project manager, the source editor and the compiler of the system are not hard-coded entities in it. All these modules are components that are assembled at start-up and placed on the application's main Shell, collaborating in order to present a homogeneous, solid environment.

Delta is a compiled, dynamic, object-oriented language developed internally in the Institute of Computer Science, Foundation for Research and Technology – Hellas (ICS-FORTH). It is classified among the dynamic object-based languages (2). Object-based programming is a style of object-oriented programming where classes are not present and behavior reuse is performed via the process of cloning existing object instances (3). Delta is a compiled language, meaning that a program is compiled into byte code. The machine-independent byte code can then be loaded and run in the Delta Virtual Machine (VM).

Dynamic scripting languages rapidly gain an important role in the software development cycle and are complementing traditional system languages like C/C++ and Java (4). Such languages are usually considered higher-level languages that provide strong constructs and machine abstractions to the programmer to efficiently write programs. They alleviate the hassles of having to put up with low-level programming details, at the cost

of performance efficiency. Fortunately computers increasingly improve their performance and computation power, thus this trade-off is no longer an important issue when compared to potentials offered for code reuse and systems' extensibility by dynamic languages.

Such languages promote polymorphic meta-programming, authoring of generic functions that can be applied on different data types. It is analogous to C++ template meta-programming (5), though Delta is an un-typed language with no need of the template construct. In contrast to the most conventional languages where types of objects are available at compile-time, in object-based languages most programming constructs are performed through language design patterns, like factory functions for construction of new objects instances.

1.1.2 Circularity dimension

The circularity property of Sparrow is attributed to the fact that programs developed inside the environment, can be integrated to the same environment, extending its functionality and features. Major requirement of this feature is the ability of Delta programs to interfere with the IDE and manipulate its components. Furthermore the environment should be able to call Delta functions to allow script extensions running inside it.

Most wide spread IDEs offer circular extensibility of the system in some form. However, they imply familiarization with a complicated process that most users are not accustomed or are not eager to take up. A friendlier, well-supported approach is required in order to make the programmer comfortable to write her own extensions. Extensions are a major feature in our environment, and our design has been evolved around this notion.

The supported ways for third parties to extend Sparrow are loading of dynamic libraries and authoring of Delta scripts. A precompiled dynamic library, which implements environment's tools, can be loaded at runtime. It can register in the system, becoming capable to control and interfere with other component providing its facilities. The extension programmer has also the option to manipulate the environment through scripting. Scripts that are developed inside the environment can be attached to it in order to control its modules or install new ones that operate in a similar manner to those already provided. Figure 1-1 illustrates the concept of circularity in our environment.

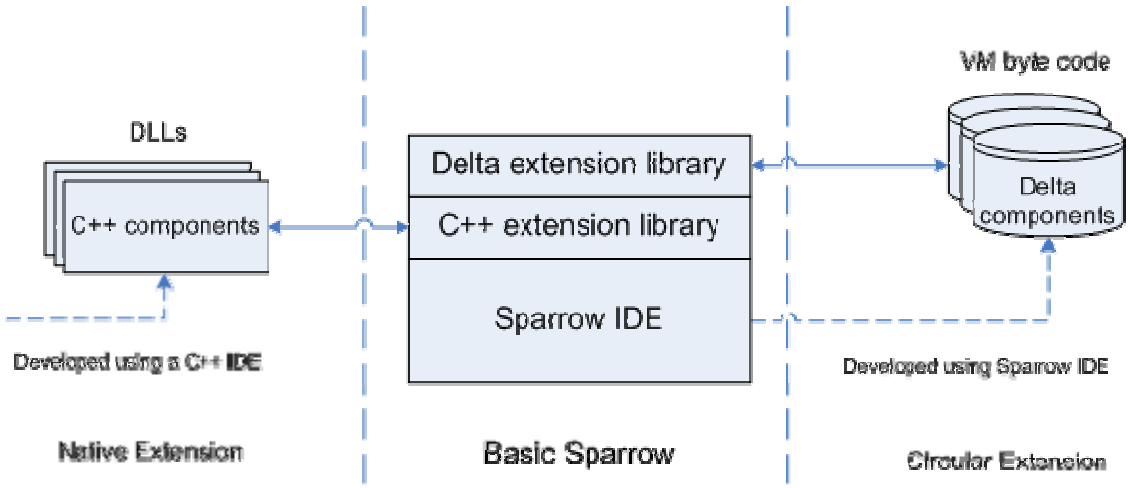

Figure 1-1 Circular IDE architecture

The basic Sparrow IDE provides a C++ extension library for native extensions, Dynamically Loaded Libraries (DLLs) implemented in C++, to operate inside the environment. On top of the C++ library a Delta extension library is build facilitating virtual machine binary code, programs implemented in Delta, to operate inside Sparrow IDE. The Delta programs are being developed inside the environment, clearly illustrating the circular attribute of the system.

1.1.3 Meta Dimension

The "meta" notion is used to represent a general solution to a problem domain by providing a model, which can be specialized to generate precise solutions for particular problem instantiations. Sparrow is characterized as a meta-IDE due to the fact that it is not constrained to a single solution for providing an IDE for the Delta language. Instead, we deliver an environment platform that provides a core IDE, assembled by related components, which can be extended and customized to deliver distinct IDE instances. Sparrow offers a tabula rasa concept supplying a basic application, where desired components can be built and attached to a modular infrastructure as extensions.

The "meta" property of Sparrow is attributed to support for extensions, adaptations and deployment. We already discussed circularity support in Sparrow under section 1.1.2, and explained its extension support.

Horizontal and Vertical Extensions

Addition of new components, which extend functionality of core components by manipulation, is described as *horizontal extension* of the system. Sparrow IDE also allows installed components to be deactivated or replaced by others, defining new internal behavior. Two constraints are imposed to replacing components in order for the environment to remain functional. They should obey the original API, *runtime consistency*, and they should provide related semantic behavior, *usage consistency*.

Dynamic substitution of components is *vertical extension* of the system. The environment, thus, customizes itself to the programmer's needs dynamically during runtime. The adaptation in the system is provided by two distinct methods; configuration, based on decision making rules and dynamic assembly of components, based on scripting. However, the adaptation processes are not fixed in the platform. We will discuss in a later section how they can be extended.

While horizontal extensions build on top of already provided tools, the vertical extensions drop supplied functionality to replace it with components and tools that meet the required needs. Both extension mechanisms imply the capability to construct domain specific environments using Sparrow IDE. These environments target a narrower domain than the broad programming domain, allowing development of tools that further automate and aid their development processes.

Sparrow facilitates open deployment by third party tools to build domain-customized development environments for application systems that rely on the Delta language. It reflects the *tabula rasa* concept via architectural openness. It provides an open component infrastructure, offering fundamental components and emphasizes the incorporation of additional functionality through the development of extension components.

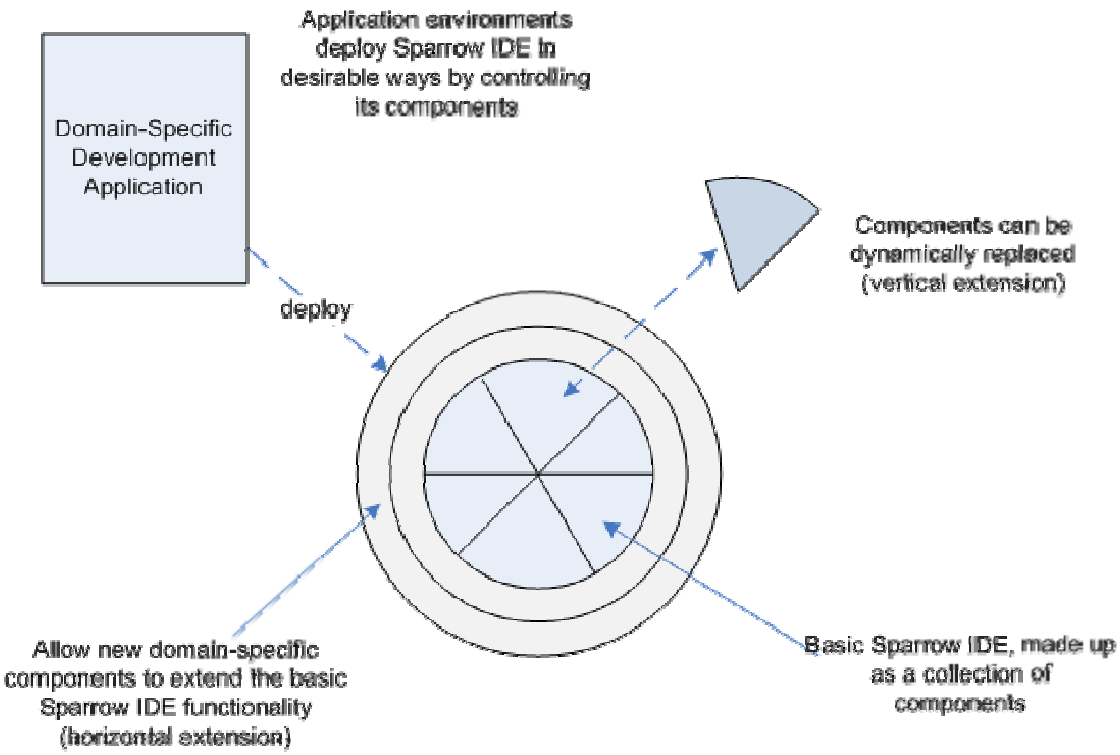

Figure 1-2 Horizontal and vertical IDE extensions

Figure 1-2 gives a visual representation of the horizontal and vertical extensions of the environment. Sparrow IDE is visualized as a small core composed of a few basic components. *Horizontal extensions* introduce new functionality by building around the provided core, for instance call graph visualisers or documentation generators. *Vertical extensions* replace component with alternative versions which provide similar functionality or extending the already provided. Horizontal and vertical extension mechanisms allow third-party applications to deploy Sparrow IDE in their development process by controlling components of the environment.

Domain Specific IDEs

Domain specific IDEs are considered the environments that accumulate tools, similarly to IDEs for programming languages, though they are specific for a job in a more strictly defined development domain. Circularity, dynamic adaptation and the meta characteristic of Sparrow are the three key ingredients of our system that allow the construction of domain specific IDEs.

An example of a domain specific IDE is a game editor. Game editors are applications that allow the construction of games for a specific game engine. Game engines are usually complex systems that take care of the low level details of a game's presentation and use scripting languages for the description of the game logic in their worlds. Thus a game editor is required to organize the source script files. Instead of replicating the effort to write a sophisticated source editor, which provides syntax highlighting, code completion and source level debugging, it can remotely control Sparrow, integrating it to its development process. In this fashion, the editor can painlessly access, already existing facilities for its target scripting language.

A game editor thus aggregates several tools in a single homogeneous environment. Some of these are specific to the domain of game development, such as sprite/texture

editors, level editors, world viewers. Other components, like source editors, compilers, debuggers, build-systems, are general purpose IDE components. Our platform can provide the ability to third party tools to host the environment and incorporate those elements in their development process.

A third feature for incorporation of domain specific environments is provided in Sparrow IDE. We supply to third-parties a remote control API, deployment API, which enables external processes to manipulate the environment. This mechanism allows Sparrow to be deployed, in external application environments. External processes can then control system's components as they see fit to their requirements. Deployment of Sparrow to external applications permits control of all system modules as if the application was a component of the environment. Remote deployment allows external extensions of Sparrow to construct a Domain Specific IDE, in contrast to the internal construction facilitated by the extension and dynamic adaptation mechanism.

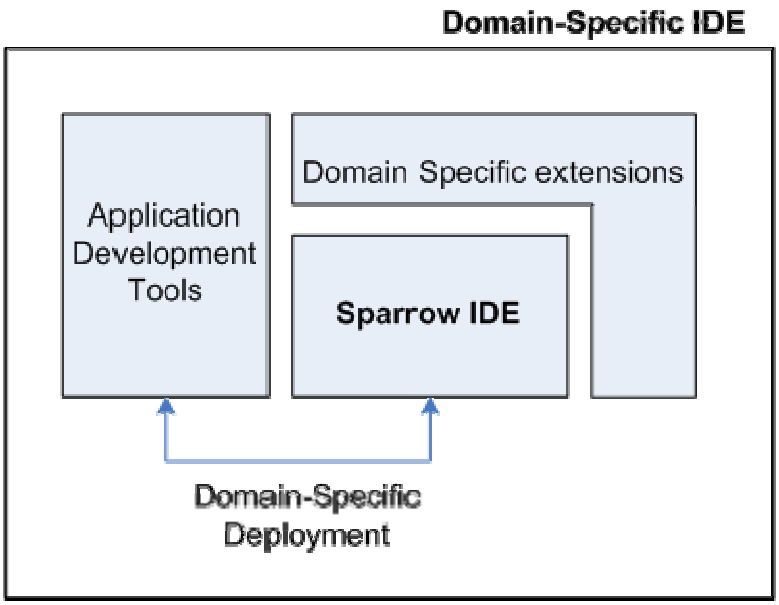

Figure 1-3 Meta IDE architecture

Figure 1-3 gives the architectural view of a domain specific development environment that deploys Sparrow IDE. The external application controls Sparrow IDE through its deployment API, being able to adapt it to its requirements. Adaptation may include installation of domain specific extensions in the environment. Those extensions can be implemented either natively as DLLs or in Delta, using the environment.

The communication between the two processes happens over a network protocol. As a result the host application should have a module that implements our protocol's stack in order to remotely control Sparrow and manipulate its scripts. This extension, depending on the host application, can be added via a plug-in, possibly a dynamic library, or by rebuilding the host system, adding the extension code to its core source base. The two applications as a whole, the external environment and deployed Sparrow IDE, are considered as a new domain specific IDE.

Software Frameworks

The "meta" property of Sparrow seems to resemble software frameworks. A framework describes a reusable software design, expressed as a set of abstract components (classes). Families of related applications are defined by a skeleton where code (components) fills in the gaps to define a specific implementation. The skeleton provides hooks for component interaction with the overall system. A representative example of work on software frameworks is Unidraw (6), a framework for graphical editor applications.

A framework provides flexible software modules with high degree of parameterization for customized components to reuse, avoiding shared code rewrite among similar classes of applications. A framework alone is not an out-of-the-shelf application, but rather a template that needs to fill in its gaps in order to produce a final application. These gaps are filled by customized components in addition to parameterization input resulting to a discrete solution for the problem. This is the main distinction between

Sparrow as a meta-platform and frameworks. Sparrow is already a usable application with no need for definition of additional components in order to be operational. Instead it allows manipulation and substitution of its already provided components as well as addition of new ones to extend it and customize it to a domain specific application.

The two paradigms may effectively be combined, and their fusion, as implemented in Sparrow, gives a meta-IDE whose components are essentially framework components that encompass a skeleton for future components to fit in. This prescribes hierarchical component activation during start-up, requiring at the implementation-level loose coupling among components.

1.2 Primary Objectives

The objective of this Master's Thesis was the development of an IDE for Delta that supports:

1. feature extensibility via scripting (circularity), by third parties
2. an interactive source level debugger
3. dynamic, adaptation-oriented assembly of the environment based on user profiles
4. an interactive project manager

One matter that must be clarified is the reasons behind our approach to develop an IDE platform from scratch, instead of extending an already existing industrial strength solution in order to implement an IDE for Delta. There are popular IDEs, which allow hosting of new languages via extensions. Programmers may plug-in their compilers, editors and debuggers to the environment's hooks and rely on the system's infrastructure for their interoperability.

The latter option, extending an existing IDE, is a viable solution that would allow rapid integration of the Delta language to a widely spread environment. Nevertheless our intention was to push the state-of-the-art in the environment platform technologies, an

objective that cannot be achieved by reusing an already existing application. Our proposal expands the features available to currently available IDEs.

Below we will briefly discuss the main axes of our work. They clearly support our decision to develop a new IDE platform instead of extending an already existing one to integrate Delta. These properties will be analyzed in more depth in following sections.

1.2.1 Extensibility Layer for Delta

The extensibility of Sparrow via scripting supports three application classes for Delta scripts:

- manipulation of available components
- configuration of the layout and behavior of the environment
- creation of new components and their integration to Sparrow

The second role can be considered, technically, similar to the first one as layout configuration of the environment entails the manipulation of the core's Shell component, by specifying the instantiation and positioning of other add-on components in it. Nevertheless it is identified as a second application due to the uniqueness of the Shell component as well as the significance of the environment's layout specification. Consequently an extensibility layer for Delta is needed for the IDE in order to facilitate these applications.

The first two roles can be supported by offering library functions for Delta that can manipulate the environment. Library functions are native C/C++ functions that provide an interface to the scripting language to interact with the system. The implementation of library functions is non-trivial and raising several issues including translation of basic data types as well as user-defined data types (structs/classes) from one language to the other.

Moreover it is important to provide a reasonably small function set, which in the same time will be generic enough to describe every environment's component. If the library set was bound to specific components then every third-party should re-implement a library for its components in order to be accessible in Delta. This is undoubtedly an irksome and error-prone process.

A generic library set is provided which is not bound to the core components of Sparrow, thus supporting third-party components out-of-the-shelf as well. The creation of such a library is assisted by the component's reflection infrastructure of our system. Due to the constraint of providing a generic library, the function set seems somewhat complicated and unnatural to the Delta programmer. Thus, we further provide a library written in Delta build on top of the provided library functions in order to wrap their functionality and supply an API that is closer and more suitable to the Delta programmers.

For support of the third class of applications, the creation and registration of components, we provide another library function set, which allows a script to define components that can be loaded and used inside the IDE. Furthermore we provide an integrated tool that lets the programmer to effectively manage her extensions, supporting the authoring of such scripts.

1.2.2 Source Level Debugger

Programmers have been accustomed to debug programs inside their development environment and expect an interactive, graphical, source level debugger from every IDE. We provide an interactive debugger for the Delta language that is built as a set of component in our IDE. The debugger kit for the Delta provides the necessary tools that are typically encountered in modern environments. More specifically there is a graphical breakpoint manager in addition to marking of breakpoints on the source editor, an interactive call stack view of a program at breakpoints and a window of expression

watches. On top of those tools we build extensions, implemented in Delta, which further help the debugging process.

As the programmer is able to write Delta scripts that manipulate and extend Sparrow, it is naturally expected to be able debug those scripts inside the environment as well. Evidently a programming error (bug), in such scripts, could have as a result to bring the environment to an erroneous or inconsistent state. Alternatively the script that is running in the debug mode may interfere with the debugging facilities. These effects can confuse the programmer, who would not expect an unexpected behavior from her system or even have the system to halt functioning due to a bug. In order to avoid such situations, the environment spawns a new Sparrow instance and attaches/debugs scripts that manipulate the IDE in this sandbox. The effects of the script are applied on the newly instantiated IDE, while the programmer can work on the former unaffected by the changes.

1.2.3 Runtime Adaptation Infrastructure

We wish to provide dynamic adaptation of the environment to the programmer's needs based on user profiles. Users of an IDE, being programmers, are considered advanced users, thus we expect them to be able and keen to shape their environment according to their needs, provided there is a strong infrastructure that can support such functionality.

There are, currently, two alternatives in the system for configuration of the environment and the behavior of components. The first one is the manipulation of the IDE through Delta scripting as it has been already mentioned. The second is a configuration based on Decision Making Specification Language (DMSL) (7), which will be further analyzed in a following chapter.

The main facility for profile configuration in the IDE is the Adaptation Manager, a component of the system itself. The Adaptation Manager is a framework component for the configuration of the system. It defines the notion of a profile but does not define how the configuration is done. It provides the hooks for components that will be able configure the system to be attached and apply their configuration logic. The two configuration alternatives already mentioned, are components plugged in the configuration platform, instead of being a hard-coded part of the manager itself.

User profiles do not only affect the behavior of components in the environment. Such a mechanism would be considered as customization/configuration of the system. The adaptation of the environment implies dynamic substitution of components. The results of the dynamic assembly may vary from slightly modified to distinct IDEs with different capabilities. Programmers are given the control to craft an environment they can feel comfortable with.

1.2.4 Interactive Project Manager

The Project Management is also another important aspect of an environment. It also implemented as a framework component to our system, similar to the Adaptation Manager. The project management consists of the mechanisms that define the relations between compiled units in an application. It manages the structural organization of an application source files and their build system.

The framework component defines the conceptual idea of the project management and leaves placeholders for other components to fill and implement the notion of a project and a source file, as well as the concept of building them. The Project Manager, that has been developed, is independent of the language and its compiler. It defines the skeleton functionality leaving well specified gaps for other component to be attached and implement the appropriate build process for every compiled unit. As a result the

project's organization is completely autonomous and decoupled from the compilation process.

1.3 Sparrow IDE Macro Architecture

The architecture of Sparrow is component-oriented, meaning that all of its parts are implemented as self-organized, autonomous modules. The macro architecture of the environment is presented in Figure 1-4. There is top level container of environment's tools, the User Interface Shell. Components are installed inside the Sparrow shell bringing their functionality to the environment. The most notable components are presented in figure, including the Project Manager, Source Editor, Debugger, Adaptation Manager and Delta Wrapper. The Delta Wrapper is the component that embeds the Delta language in Sparrow interfacing with the compiler, the virtual machine and the debug server of the language.

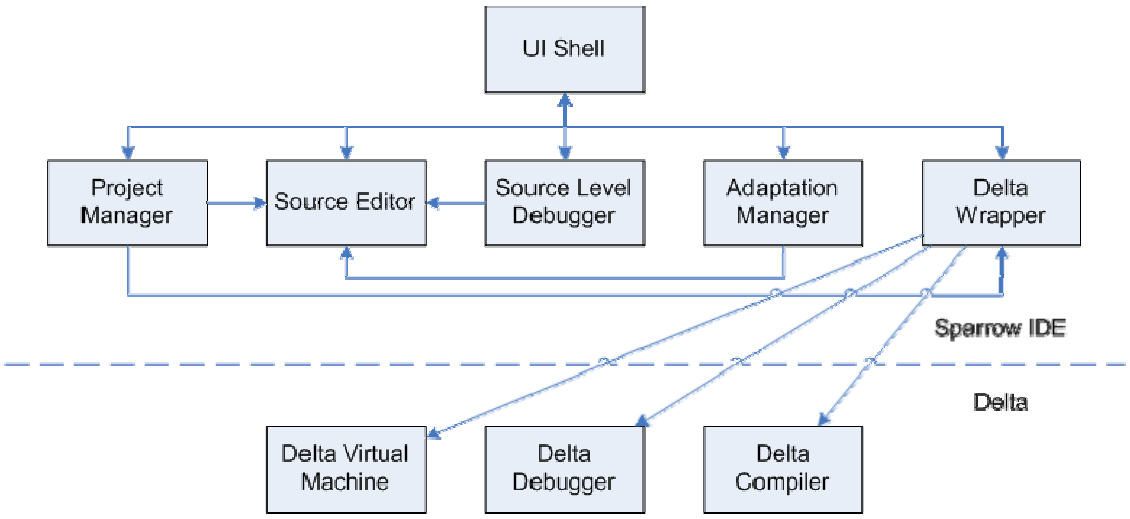

Figure 1-4 Sparrow Core Macro Architecture

The project manager is the only component of the environment that accesses the Delta Wrapper for building and debugging of programs. Hence, the environment has the least possible dependencies with the Delta environment, allowing it to be an IDE that can

incorporate more languages. The architecture also shows that most tools of tend to be built around the source editor which is the central component of interest for developers during programming.

In Figure 1-5 a snapshot of Sparrow IDE is presented with several component tools placed in the docking areas of the user interface Shell.

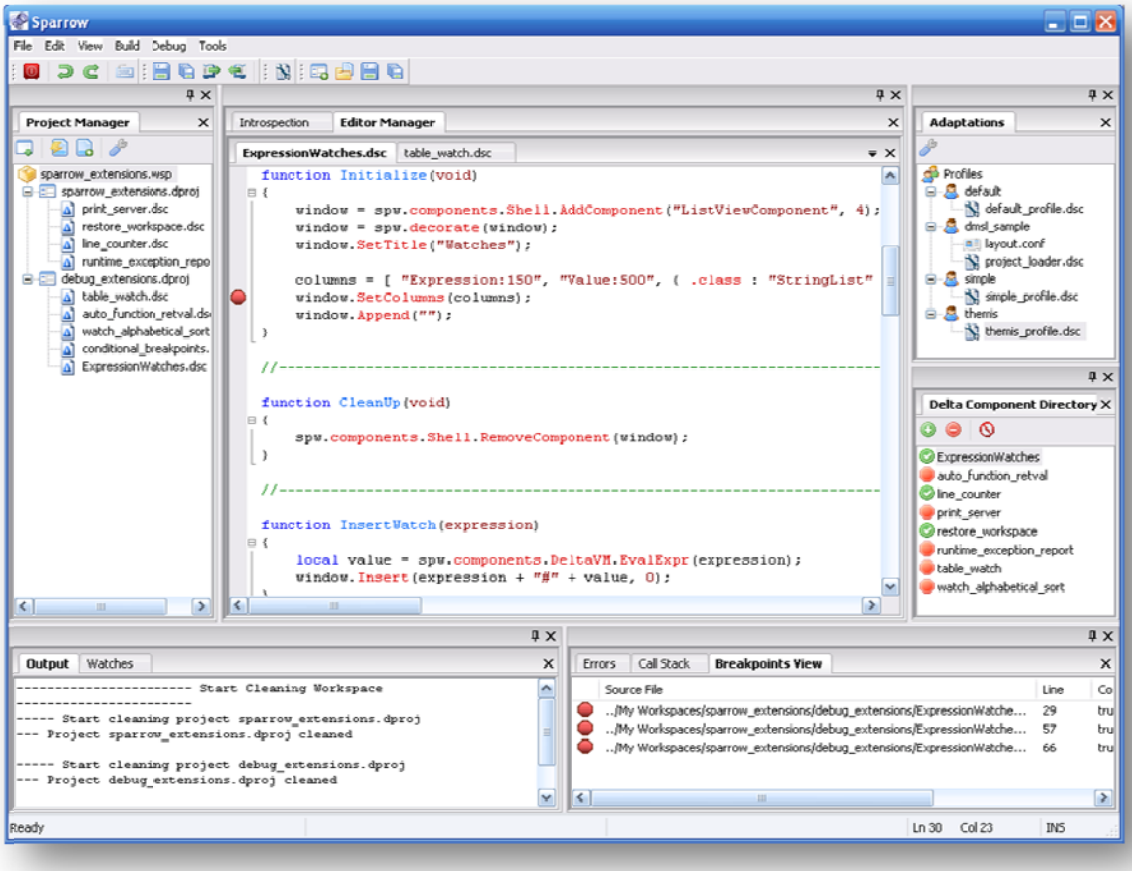

Figure 1-5 Snapshot of Sparrow IDE

Figure 1-6 presents the design of Sparrow's core infrastructure for component support. Components are registered, at runtime, to the system in a central registry. The registry holds information about the component's properties, its exported functions and some

metadata presenting a short description of its functionality. Components are identified by a unique string identifier.

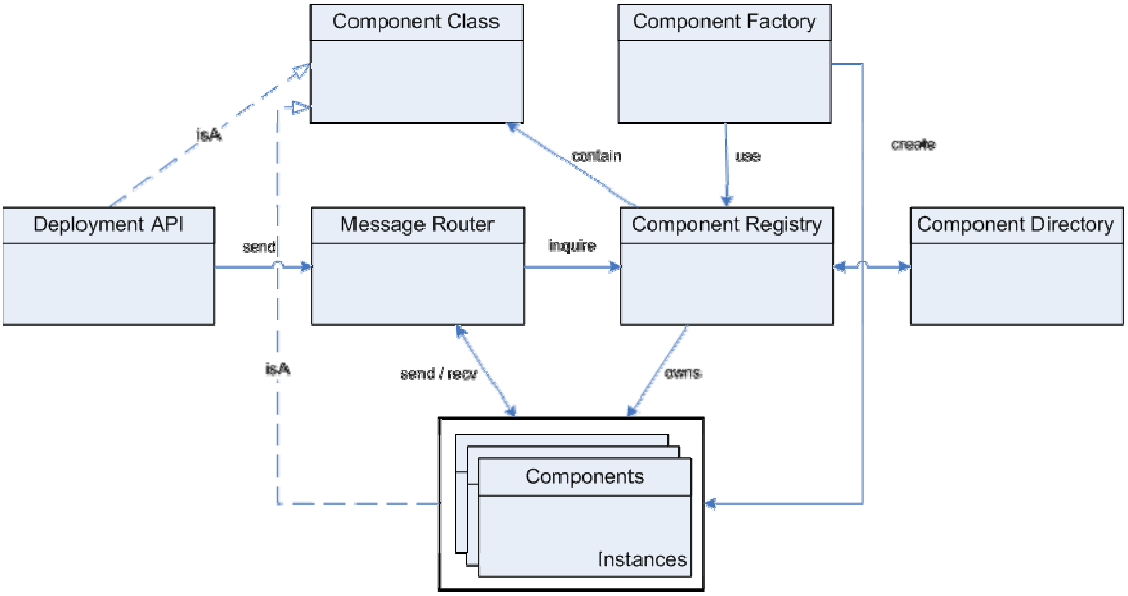

Figure 1-6 Software design for component support

Their registry entries also contain a pair of construction and destruction processes for creation of instances. Introspection is provided for registered components, allowing other components to access exported functions with their signatures and documentation, properties as well as inheritance information.

For construction or destruction of instances a component factory is used. The component factory is based on the object factory design pattern for construction of object instances (8). It accesses the main registry in order to obtain the constructor and destructor functions of a component. Instances can be constructed by other components in the system, so they are able to access the factory. Instantiated components are hold inside the registry along with their component class specification. Instances are identified by a pair of class identifier, the component's class string identifier, and a monotonically increasing serial number.

Components are loaded and registered on-demand at runtime by the component directory. Each component is described by a definition file stored in a predetermined Sparrow directory. A definition file specifies the type of the component and its physical location. Currently three types of components are supported in the environment; build-in core components, components in native code loaded as dynamic libraries and script component written in Delta. The definition files are loaded during bootstrapping by the directory. It is possible to support additional component types by plugging a new module in the component directory. For each component type a loading and unloading process should be provided, which is responsible for locating and registering components of that type in the registry. When a non-available component, in the registry, is referenced in the environment, the directory is notified and looks it up to its catalog. If the component's definition is successfully found, it is loaded, according to its type, and registered in the system.

The message router in the architecture is the module that allows communication between component instances. Communication between components in the system happens in the form of function calls, though the components are not connected at linking-time. An underlying mechanism based on message exchange between component instances allows the function call abstraction. The message router is responsible for dispatching of messages to the appropriate instance. It accesses the component registry where type information of functions and available instances reside and forwards the message to its corresponding destination.

Script components communicate with the environment in a similar manner. However a script component resides inside a Delta Virtual Machine and is not able to interfere with Sparrow's message routing mechanism. Thus a Delta script proxy wraps a script component to bridge the worlds of native C++ code and Delta scripts. The white-headed arrow in the illustration denotes the fact that the script proxy is a component class itself, allowing a script to access all the infrastructure mechanism as if it was a

component written in C++. Scripts control the proxy via Delta library functions. The double-headed arrow between the Delta proxy and the message router emphasizes the fact that messages are dispatched from and towards a script component.

A similar approach is used for supporting external host applications via the deployment API. It is constructed as a component in the system as is the case with the Delta proxy. However, the one-way arrow to the message router signifies the fact that messages travel only, from the deployment API to the rest components. The reverse path is not acceptable as an external application would not accept requests from the environment. The different colored modules of the architecture highlight the fact that those parts are external to Sparrow and communicate with it through provided gateways.

2. Related Work

2.1 Extensibility of Software Systems by Third Parties

Scripting languages most often are used for extending and customizing large-scale platforms written in traditional system languages. They are rarely used for creation of large systems from scratch. Examples of systems that can benefit from scripting extensions are modeling and CAD applications, game engines and web browsers. A system can support extensions via a scripting language by exposing hooks for external control. For each language the system wants to incorporate, it should expose library functions that allow control of the environment.

An Integrated Development Environment is a substantially large-scale system that substantiates the use of a scripting for extensions. Ideally scripting in an IDE would allow developers to control their environment and extend it by writing add-ons and tools that meet their custom requirements.

It also commonly encountered, for applications that need to manipulate or edit script, to internally implement modules of a development environment. Such approaches suffer from either an expensive development circle, duplicating the effort on tool development, or provision of incomplete solutions.

A more appropriate solution, nevertheless, would be to provide a generic platform, open to extensions, which would allow application that need script editing to deploy it in their development processes. Deployment implies remote control of the platform by an external application. Sparrow tries to fill the gap of the deployable platform which can be controlled to manage the scripts of an external application. Though some development environments provide an API for remote control, through OLE Automation (9), to our knowledge there is no system that utilizes those APIs in order to integrate

these environments to an external development process. Instead most attempts for extending are focused on embedding specialized tools inside the environment, using the provided plug-in framework of each system.

2.2 Dynamic Adaptation-Oriented Assembly of Components

Dynamic adaptation of software on user needs has been noted in previous studies (10). Adaptation entails customization of the environment to the user's needs based on information provided by profiling. Adaptation of an environment by assembling components based on user profiles implies decomposition to modules with discrete roles in the system. Furthermore it suggests provision of interchangeable components and hierarchical activation during start-up. As stated in (11), the key architectural implication, due to the functional requirement for dynamic assembly, is the need for organizing software components in order to enable dynamic architectural containment hierarchies.

The requirements posed in a system, lead by necessity to careful dissemination of functional roles, thus to clean architecture design, which would allow extensions to comply with future requirements. Another side-effect of this approach is the loose dependencies between components, as they are programmed to work independently, even in possible absence of a component. Hard links between components are eliminated to allow assembly during runtime.

Sparrow's component-oriented architecture was driven by the engineering requirements for clean design and specification of roles for adoption to several scenarios. To meet those criteria, our design was leaded to an architecture similar to this posed by dynamic adaptation. Consequently the adoption of dynamic adaptation in our environment was a natural step for further pushing customization and extensibility.

A key factor for the adoption of this feature, from the users, is the provided means for describing their requirements and constraints for the system. A usual approach is the use of configuration files and provision of graphical interfaces that assist users on editing. However configuration files can only give a static description of the environment assembly, as they are unable to apply any form of logic to the configuration. In our approach we expand configuration options using a domain specific decision making language as well as scripting logic.

The specification language we use is the Decision-Making Specification Language (DMSL) (7). DMSL is a domain specific rule-based language for making decisions on activation / deactivation of components during the run-time of a system. Sparrow incorporates a DMSL compiler for extraction of decision based on supplied user profiles. The basic constructs of the language are if..else blocks that describe the adaptation logic allowing the compiler to extract decision based on properties described in configuration files.

2.3 Component Oriented Architecture

The functional requirements of the system dictate the approach of a component-oriented architecture (12). In component architectures software is developed by gluing prefabricated independent software modules together, namely the components. The immediate benefits from software components are reuse of code, encapsulation and composition with other components.

Components are interoperating via well defined interfaces that expose their functionalities to external collaborators. The separation of components represents their functional role in the system. This approach further pushes the principles of Object-Oriented Programming where functional roles are decoupled into different entities (13).

The borders among components are lucid, clearly underlining the loose inter-component dependencies through their exported API. They are a building block for an

infrastructure that provides a system capable of adapting during use by instantiating variants of a module that serves a particular role in the system. Parts of the application can be dynamically assembled at run-time rather than being statically associated within the source-code of the application, which allows the system to be orchestrated during use, in contrast to the monolithic approaches of many other systems.

2.3.1 Component Communication

Components need to exchange information to cooperate and compose a functional application. The need for a communication mechanism originates from the fact that components are not linked statically together and are unaware of the existence of each other. Furthermore interfering components may not even reside in the same memory address or even machine.

Communication among components relies on message exchange, instead of calls that jump on specified memory addresses that are bound during linking-time. This communication model allows a broader collection of entities to be used as components. For instance components may not be necessarily written in the same programming language. It is even possible for a component to be a remote process that communicates with the system over a network.

Message exchange is not trivial and entails many details that distract programmers from their work. The abstraction of function/procedure calls among components is preferred as a communication pattern. The semantics of procedure-calls are a notion that programmers of imperative programming languages are well accustomed with. Remote Procedure Call (RPC) is a technology that allows programs to call a function/procedure that will execute in another address space by omitting the gruesome details of remote interaction. The middleware that is used to support this facility usually hides the difficulties of Inter-Process Communication (IPC). Several middleware solutions exist for component communication in the literature like CORBA (14) and Microsoft's COM (15).

However they have a very broad-spectrum, imbuing a heavy overhead for our narrower domain problem. For this reason we have implemented a custom mechanism for communication of components and dispatching of messages.

2.4 Overview of available IDEs

Implementation and maintenance of development environments is a demanding task. Most industrial strength IDEs available, have large development and support teams that constantly develop them. Two of the most wide-spread and representative environments are Microsoft's Visual Studio (16) and the open-source Eclipse platform (17).

Both environments target several programming languages and provide means for the programmers to extend them. Especially in Eclipse there is a broad range of extensions varying from support for new languages to customized tools.

Visual Studio supports several mean for environment's extension varying in difficulty and potentials. The simplest extension practice is VSMacros that are actually recording of user action in the UI and saving them as macros in the application. More interesting extensions are Visual Studio's Add-ins, which are dynamic libraries following the COM architecture for integration in the environment. They can be authored in any language that supports COM, which translates to virtually every language that is supported in Visual Studio including C++, C# and VB .NET. Add-ins can implement new functionality and tools in the environment. A rich API is provided for the environment's control and the process is well supported from the environment, providing template projects for add-ins. Add-ins can be enhanced further with wizards, which are in essence step-based GUIs for completion of a complex task, typically creation of new project templates. Figure 2-1 illustrates the Visual Studio environment and the project template for creation of add-ins for the environment.

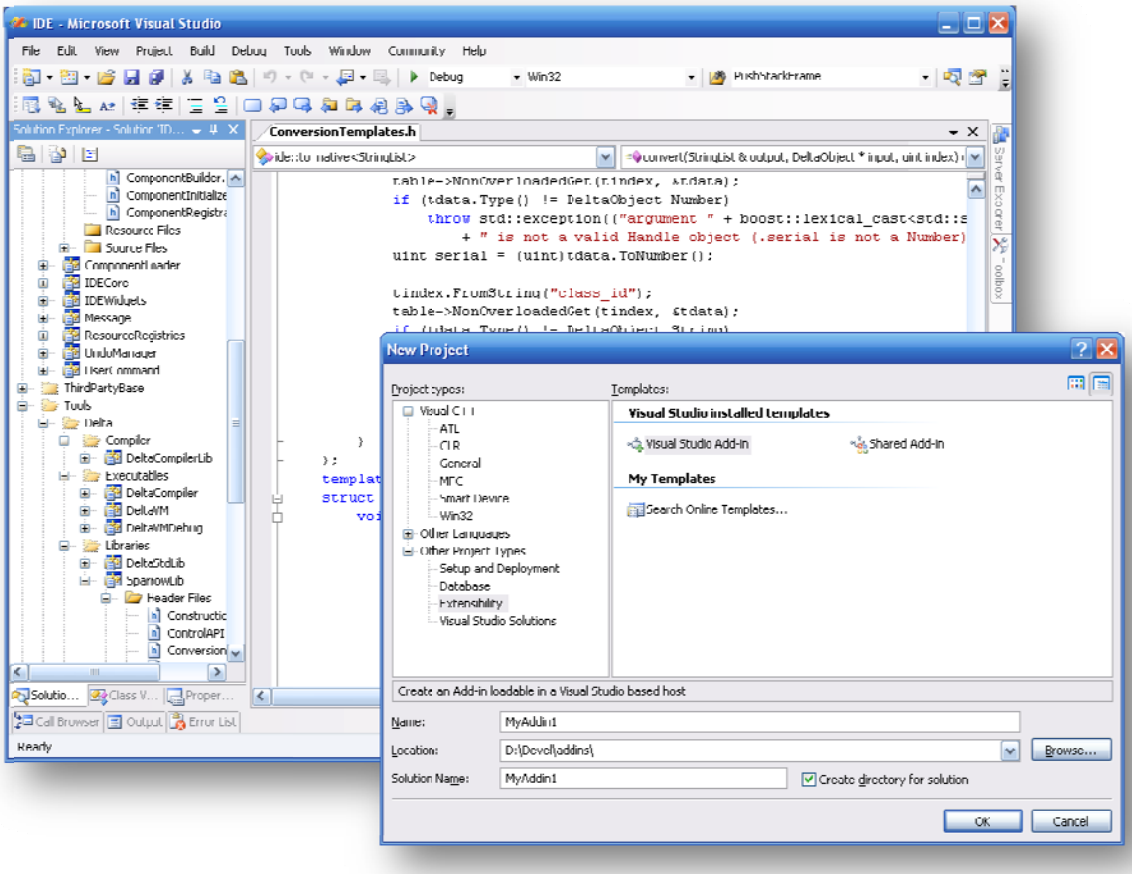

Figure 2-1 Microsoft Visual Studio environment and creation of Add-in project dialog

Finally there are VsPackages that are the most powerful type extension as they allow implementation of new editors, designers and compilers for a language and their integration in the environment. Actually all languages that are incorporated in Visual Studio are developed as VsPackages, a collection of tools for the environment that support the development in a specific language. VsPackages, like add-ins, implement a COM interface and need to be registered in the environment.

Visual Studio also offers OLE Automation (9) which resembles our goal for external deployment of a development environment. OLE objects are actually COM objects that implement the IDispatch interface, which allows a client application to find out what

properties and methods are supported by an object at runtime. However OLE is not well supported in Visual Studio as its object model is not documented and the provided API should be extracted by the programmer using external tools, browsing the provided functions.

The Eclipse platform itself is structured as subsystems which are implemented in one or more plug-ins. The subsystems are built on top of a small runtime engine. The desktop development environment on top of Eclipse is referred as Workbench. The Workbench aims to seamless tool integration by providing a common paradigm for the creation, management, and navigation of workspace resources.

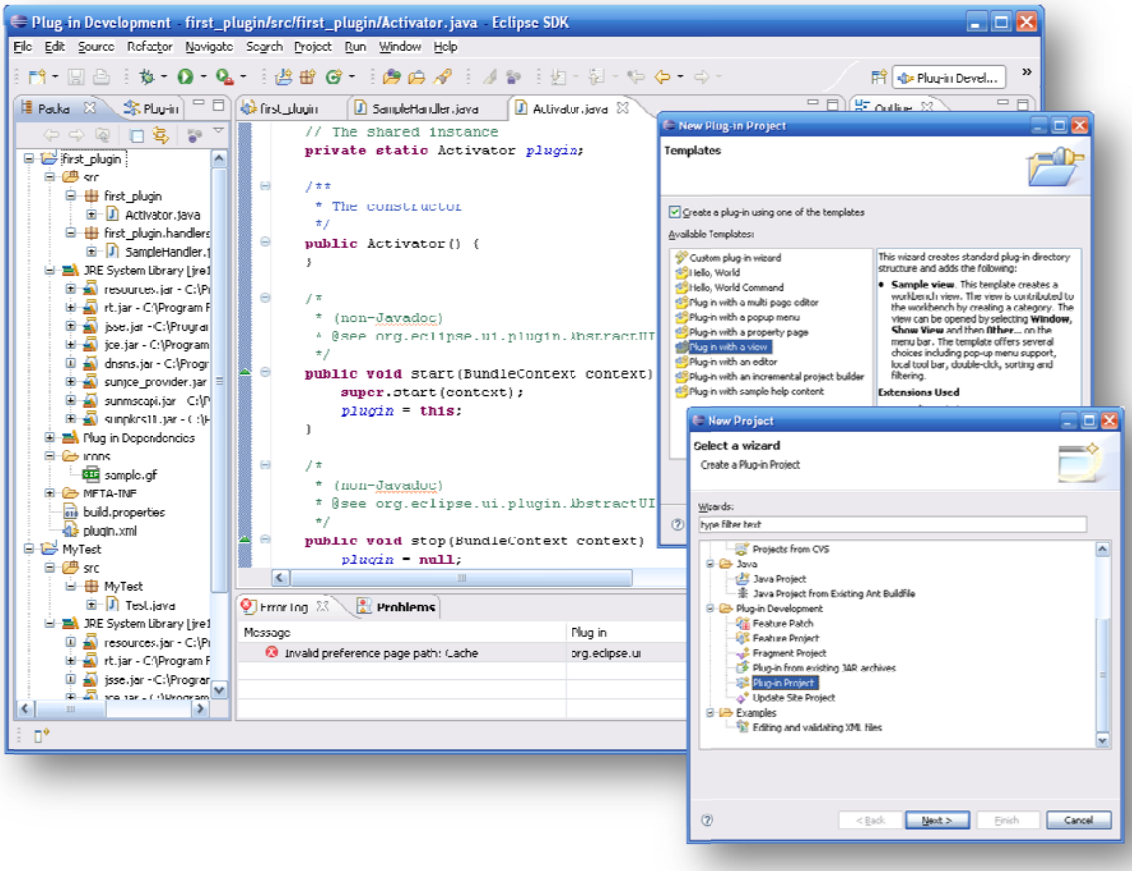

Figure 2-2 Eclipse environment and creation of Plug-in project dialog

Figure 2-2 gives an impression of the development environment, presenting the main workbench and wizard dialogs that construct template projects for creation of plug-ins for the environment. Workbench windows contain one or more perspectives. Perspectives are configurations that contain views and editors. They also control what appears in menus and tool bars.

Eclipse supports extensions in a similar fashion to Visual Studio, providing a rich development kit for plug-in authors to use. Furthermore it offers presets of templates that aid the programmer to painlessly create her extensions, automating creation of boiler-plate code through the use of wizards.

3. Extensibility Layer for the Delta Language

3.1 Technical Approach

As mentioned in 1.2.1 the circularity of the environment permits three applications:

- manipulation of available components
- configuration of the layout and behavior of the environment
- creation of new components, attaching them to the IDE

The first two applications are technically similar requiring control of components from the Delta code. This can be achieved by providing a library function set that allows access to the registry of components in our system and bridges Delta programs with the communication mechanism of the platform's components.

An extended library function set is provided, which allows creation and registration of components in the main component registry, to facilitate the requirements posed by the third application domain. This library set expose to the scripts the necessary facilities of the component infrastructure of the platform, including component inheritance, definition of user commands and authoring of component meta-data.

Furthermore a graphical component tool is provided that aids the authoring and organization of components written in Delta. Each component in the system requires a definition file that is expected to be provided from its author. The Delta Component Directory supports the authoring of script components by automatic the registration process for the programmer and managing the updates of scripts.

Sparrow runs internally a Delta virtual machine for execution of scripts that operate inside the environment. During the startup of the virtual machine a main program script

is executed that pre-loads libraries supplied by the environment to the third-party script authors facilitating composition of scripts. Such libraries will be described in later sections. Most environment scripts will use the main script's virtual machine to use the offered libraries, though it is not necessary for the programmer.

Though Delta is a centerpiece of the Sparrow environment, it is not included as a monolithic part of the environment. The compiler and virtual machine are attached to the system as components much like everything else. The design decision of decoupling the language from the environment leads to a more powerful, flexible system further promoting code reuse. A third party can easily integrate another programming language in Sparrow without the need to re-implement the build system or the source editor.

In order, for a programmer, to support another language, she should author two components wrapping the respective compiler and run-time system. The components should follow the same interface the Delta language components follow. The interface is described in section 3.6 and is abstract enough to support other programming language. The alternative language components can be loaded in place of the Delta compiler and run-time system (the virtual machine) changing instantly the IDE's target language. The dynamic substitution of components of the environment is the basis for dynamic adaptation as it will be presented in chapter 5.

The compiler and run-time components follow a certain policy describing a concept. A concept in this context is the provision of a set of exported functions that can be called by other components. For instance, a compiler component implements the compilation concept. A concept does not only describe the functionality of a module in the system, it also defines the interface that should be provided by a component in order to be compatible with it. There is no constraint so as to what the compiler component is, on the condition that it provides specific functions that can be used by the build system of

the project manager. The same rule applies to the run-time system of the language, as well as any other component of the system.

3.2 Delta Library Functions for Extensibility

In this section the Delta library functions, provided for the extensibility of Sparrow, are presented. The function set has been developed in a fashion that allows it to be generic, in order to instantly support newly created components without requiring any special registration process. Additionally we attempt to have a small set of functions avoiding functionality overlapping.

A component in the system is being described by a handle. The handle is a pair of a string, the class id of the component, and an integral number, the serial number of the instance. The class id is the unique identifier of a component, allowing access to its record in the directory and the registry of the system. The serial number is a monotonically increasing number referring to a specific instantiation of a component. This pair can uniquely describe a component. A component instance can also be referenced only by its class id; however this is a reference to the currently focused instance in the environment.

We avoid passing direct pointers to native C++ instances of components to Delta scripts, as this could result to scripts having dangling pointers to instances, destroyed outside the Delta code. On the other hand, a handle can be validated by requesting the registry of components about its existence, thus making calls of library functions safe. When such a call fails, it throws a runtime exception.

3.2.1 Data Translation Layer

The interfacing between the data types of the two languages, C++ and Delta, is non-trivial and requires translation of objects exchanged between calls. A translation layer

has been constructed that converts Delta objects to Sparrow data types, and vice versa. The translation layer supports simple types as well as some aggregate types. Aggregate types are converted to Delta prototypes[1], whose mapped data for the string key "class" is the name of the Sparrow system's type. This is a convention in our system, used for runtime type-safety of the calls.

Extensibility layer types	Delta types
int/ uint / float / double	Number
bool	Bool
String	String

Table 3-1 Sparrow to Delta Simple Types Conversions

Table 3-1 illustrates the data translations of simple data types, while Table 3-2 contains the transformation of aggregate data types to Delta prototypes.

Extensibility layer types	Attributes	Values
Handle	"class"	"Handle"
	"class_id"	String
	"serial"	Number
StringList	"class"	"StringList"
	#i	String
HandleList	"class"	"HandleList"
	#i	Handle
UserCommandDesc	"class"	"UserCommandDesc"
	"class_id"	String
	"func_name"	String
	"isCheck"	Bool
	"flags"	Number
	"image"	String

Table 3-2 Sparrow to Delta Aggregate Types Conversions

[1] Prototypes in Delta are associative tables that map a value to a key, both of arbitrary types.

Apart from the Handle class, the *StringList*, *HandleList* and *UserCommandDesc* types are supported. *StringList* and *HandleList* are collections of *Strings* and *Handles* respectively, represented in Delta with a table whose i^{th} numbered index maps the ith stored string in the list.

UserCommandDesc is a description of a user command in the environment. The user commands represent operations of a user in the application that are presented in the main menu bar, the tool bars and the context menus of the application. It defines a handler of the command, described by a class id and a function id, which will be called when the command is activated by the user. The programmer should also supply some additional parameters to the command; defining whether this is a check command or not, where the command should be placed (menu bar, tool bar, context menu) and optionally a representation image.

3.2.2 Component Control Library Functions

In Table 3-3 the library set provided for extensibility support is presented. It includes functions for construction, destruction and access of components. Each function is presented along with its arguments and return values.

handle	**spw_createcomponent**	(class_id, {handle | class_id})
void	**spw_destroycomponent**	({handle | class_id})
void	**spw_setparent**	(handle, {handle | class_id})
handle	**spw_getcomponent**	(class_id[, serial])
class_ids	**spw_getcomponents**	(void)
void	**spw_setproperty**	({class_id | handle}, property_name, property)
property	**spw_getproperty**	({class_id | handle}, property_name)
properties	**spw_getproperties**	(void)
ret_val	**spw_call**	({handle | class_id}, function_name[, arg0, ..., argN]

Table 3-3 Sparrow's Control API

We should explain the notation of in the arguments column of the above table; arguments in brackets are optional while the curly brackets contain interchangeable arguments separated by slashes. Delta is a dynamic language meaning that there is not compile-time type checking of the arguments; however it is possible to request the type of a variable at run-time. There is no function overloading in the language, however each function may be called with more actual arguments that the formal arguments it defines. There are mechanisms in the language to validate at runtime the number and types of its actual arguments, handling overloading of types or potentially raising a runtime exception.

Function `spw_createcomponent` takes a string, the class id of a component, and optionally a handle or class id of a parent component, and creates a new instance, returning its handle to the caller. Similarly `spw_destroycomponent` takes as an argument either a handle to a component instance or a string with its class id and destroys the specified instance. At any current time there is one active (focused) instance component for each component class in the system. When a class id is an alternative argument to a handle as an argument to a library function that expects a component instance, the currently focused instance is implied, if any. Function `spw_setparent` defines the parent of a component instance. Sparrow organizes component instances in a containment hierarchy. Parent instances are notified for events from their children and allow information and event flow in a structured manner. By default when a component instance is created is orphan, having no parent. All components with a graphical subsystem in order to be rendered, should be children, either immediate or recursively, of the top level container component of the environment, the Shell.

Function `spw_getcomponent` takes as an argument a component class id and optionally a serial number and returns a handle to the requested component instance, in case it

exists. Again if no serial is specified, a handle to the focused instance is returned. A list with the class ids of all registered component is returned by `spw_getcomponents` function. The list is an indexed Delta table.

Component properties are pairs of attributes and values used for the configuration. Both primitive and aggregate types of properties are supported. A full list of provided property types can be seen in Table 3-4 along with their representation in the Delta language. They are user-defined types that are translated to Delta in the same fashion as the rest of aggregate types described in section 3.2.1.

Property	Attributes	Values
IntProperty	"class"	"IntProperty"
	"value"	Number
RealProperty	"class"	"RealProperty"
	"value"	Number
BoolProperty	"class"	"BoolProperty"
	"value"	Bool
StringProperty	"class"	"StringProperty"
	"value"	String
IntRangeProperty	"class"	"IntRangeProperty"
	"value"	Number
	"min"	Number
	"max"	Number
EnumStringProperty	"class"	"EnumStringProperty"
	"value"	Number
	"options"	String list
StringListProperty	"class"	"StringListProperty"
	"value"	String list

Table 3-4 Sparrow Properties' translation to Delta prototypes

Properties are translated as Delta prototypes, where the key "class" maps the actual property type while the key "value" maps the property's value. Apart from the apparent primitive properties, some more types are included, namely the `IntRangeProperty`, which is similar to the `IntProperty` though its bounds its value between two limits, the `EnumStringProperty`, that allows selection of a string among a predefined list of strings, and the `StringListProperty`, which is a set of strings. The String list mapped type is a numerically indexed Delta table.

Functions `spw_setproperty` and `spw_getproperty` are setting and retrieving a property's value respectively. The property name argument sets the property's label while the property should be one of the property types described in Table 3-4. `spw_getproperties` returns a numerically indexed Delta table containing the label names of a component's properties.

Finally function `spw_call` is used for calling exported functions of components. In addition to the member functions every component export, it may also export static functions. A static component function has exactly the same semantics of a static member function in a C++ or Java class, being a class member that can be called without the presence of any component instance. The first argument is either a handle or the class id of a component. A handle can be used in order to call both a member and a static function. The class id can be used in order to call a static function. If a member function is called by supplying a class id instead of a handle then the focused component instance will be used, if it is present in the environment. The second argument is the name of the function which will be called. Finally the rest of the actual arguments are translated to C++ object values and are supplied to the called component function. The supplied arguments have to comply with the signature of the collee; otherwise a runtime exception is raised. The function returns to the caller the returned value of the component's member function.

The provided library set is able to support every component in the environment even those that are loaded during runtime without requiring extensions, due to the reflection system of Sparrow. The component and function identifiers are resolved at runtime by the infrastructure of the environment, namely the component registry and the component loader. We should also note that referenced components, which are not currently loaded in the environment, are queried to the component directory of the system and loaded on demand.

3.2.3 Component Creation Library Functions

In addition to manipulation of already existing components from Delta code we want to further allow a programmer define her own components that can be registered in the environment and be manipulated in the same fashion by other components or scripts. For this feature we need a supplementary interface that will expose the functions of the environment that register components at runtime.

Table 3-5 presents the extra library functions that assist the construction of components from scripts. A component is defined by a unique string identifier in the system. Functions and user commands can be dynamically attached or detached from component definitions. Components can also inherit functionality and properties from other components, specializing or extending its behavior.

The notation in the table is similar to the notation in section 3.2.2. Function `spw_registercomponent` registers a new component to the system identified by a unique string id. The second optional argument is the class id of the base component which is derived. The derived component inherits all functions which can be overridden by the derived class to specialize its behavior. Similarly `spw_unregistercomponent` removes a component entry from the system, given its class id. `spw_setcomponentmetadata` allows the supplement of some meta-data that should

accompany a component, such as a human readable name, a brief functionality description, its author and version. All the arguments of this function are strings.

void	**spw_registercomponent**	(class_id[, base_class_id])
void	**spw_unregistercomponent**	(class_id)
void	**spw_setcomponentmetadata**	(class_id, name, description, author, version)
void	**spw_registerfunction**	(class_id, function_name, function, signature[, doc_string]
void	**spw_registercommand**	(class_id, function_name, function, command, flags[, image_id, doc_string]
void	**spw_registerslot**	(class_id, function_name, function, signature, signal_name[, doc_string]
void	**spw_registersignal**	(class_id, signal_name, signature)
void	**spw_signal**	(signal_name[, arg0, ..., argN])
void	**spw_undo**	({handle | class_id}, function_name, unod_function_name[, arg0, ..., argN])
void	**spw_unregisterfunction**	(class_id, function_name)
void	**spw_registerimage**	(image_id, source)
void	**spw_unregisterimage**	(image_id)

Table 3-5 Sparrow's Construction API for Delta

In order for populating the exporting interface of a script component functions `spw_registerfunction`, `spw_registerslot` and `spw_registercommand` are used. The programmer can register a Delta local function as an exported one. She should provide a name by which the exported function will be referenced by the outside world and also provide a pointer to the local function. Though functions in Delta are not statically typed, this is not the case for components' exported interfaces. A script component should supply the system with the signature of its exported function, describing the arguments types as well as the return type of the function. The signature description is in a string format. Optionally the script author can supply a documentation string for the function.

Function `spw_registercommand` is analogous to the function registration, however as described earlier a user command is appended to graphical environment as an operation that can be activated by the user, while still being able to call it programmatically. The signature of the user commands is "`void (void)`" meaning that they neither accept any arguments nor do they return any value, thus the programmer is not required to supply it. Most arguments are the same with `spw_registerfunction` function, though the programmer is requested to also indicate the graphical elements where the command will be available (menu bar, tool bar or context menu) as well as a command label.

```
start           ::=   content_list
                ;

content_list    ::=   SLASH content content_list
                |     SLASH content
                ;

content         ::=   priority option
                |     option
                ;

priority        ::=   L_CURLY INTEGER R_CURLY
                ;

option          ::=   SEPARATOR label SEPARATOR
                |     SEPARATOR label
                |     label SEPARATOR
                |     label
                ;

label           ::=   CHECK_SYMBOL IDENTIFIER
                |     IDENTIFIER
                ;
```

Listing 3-1 User Command micro-parser's grammar in EBNF format

The command label is a string describing the positioning of the command in a menu. Menus usually form complex tree structures with submenus. The user command incorporates a micro-parser that allows a scripter to control the layout of her menus.

Listing 3-1 illustrates the grammar of the Recursive Descending Parser that has been implemented.

Menu labels are described by identifiers, having submenus separated by slashes. At the beginning of each menu node a priority, a signed integral number, can be attached, which dictates its position in the menu. Commands with lower numbers are placed in front of commands with higher values. By default the priority value is set to zero. The double dash operator indicates the insertion of a separator either before or after the menu label. Finally the check operator indicates whether the command is a check option or not. For instance the user command string "`/{0}File/{100}--Open Workspace`" defines a user command "`Open Workspace`", which will be inserted under the menu "`File`", setting its priority to 100 and inserting a separator before the command.

Function `spw_registerslot` is also used for registering an exported function for the component. However the exported function is automatically linked with a signal in the system. Slots are functions, connected to signals, which are called when a signal is triggered, thus slots can be considered as callbacks. Components are allowed to export signals, which are described by a unique string identifier and signature that slots should meet in order to be connected. As a result, the `spw_registerslot function` requires the signal name argument in addition to the arguments of `spw_registerfunction`. This is the identifier of the signal in which the slot should be connected to. A slot should also accept as its first argument a handle which is the signal emitter, while the rest of its signature should match the one defined by the attached signal. Should the slot's signature not match the signal's exception will be thrown at run-time. The `spw_registersignal` function allows a script to export signals, specifying their signature and identifier, while `spw_signal` triggers an already registered signal, passing also the actual arguments with whom the slot will be called.

The `spw_undo` function permits the programmer to define a counter-effect action for a function that is being executed. The undo system of the environment allows the platform to revert to a previously stable state in case of an erroneous condition, by undoing all applied actions. The function's arguments are the component which registers the undo action, the name of the function that will be reverted and the function name of the undo function. The undo function is required to be a part of the exported interface for the system to be able to call it. Finally the trailing arguments of the `spw_undo` function will be supplied to the undo function when it is called.

For instance the revert action of `CloseWorkspace()` in the project manager is the `OpenWorkspace()`, supplied with the URI of the previously opened workspace. Likewise the undo function of a resource's `Rename()`, would be the `Raname()` function supplied with the resource's name, prior to the initial renaming.

The `spw_unregisterfunction` call removes a function from a component's definition whether it is a simple function or a user command. The call should be supplied with the component's id and the function id, to identify the requested function.

Functions `spw_registerimage` and `spw_unregisterimage` enable the script programmer to load an image in the environment, attaching it a unique image identifier. There is a central image registry in the environment that manages the loading and use of images, while components can refer to the resources using their corresponding image ids. The required image id is a unique resource string identifier, while the source argument of the function is the location of the resource on disk.

3.2.4 Delta Wrapper Library

The library set presented in sections 3.2.2 is procedural in nature and it does not feel in accordance to the Delta object-oriented nature. In this section we present a Delta

library that wraps the described library function, adapting them for use in an object-oriented API.

Delta Overview

Before we continue with the presentation of the wrapper Delta library, we should describe some traits of the Delta programming languages that are necessary for comprehending our solution. Delta defines object instances dynamically using associative tables. The index of a Delta table can be of any type. String indices can be accessed either via the traditional subscription notation, surrounded in brackets or with the dot '.' notation, similar to the access of class members in conventional object-oriented languages.

Listing 3-2 demonstrates a simple program, defining a Vector object, in Delta to get the reader accustomed with the language.

```
function Vector(x,y,z) {      ///< Vector object constructor
      return [
            { "x" : x }, { .y : y }, { .z : z },
            { .display : (
                  method () {
                        print("(", self.x, ",",
                              self.y, ",", self.z, ")\n");
                  }
            )},
            { + : (
                  function (left, right) {
                        return Vector(
                              left.x + right.x,
                              left.y + right.y,
                              left.z + right.z
                        );
                  }
            )}
      ];
}

vec1 = Vector(1,0,0);
vec2 = Vector(0,0,1);
(vec1 + vec2).display();  ///< overload operator + is called
```

Listing 3-2 Delta Vector program example

In the example we define a function named Vector that when called it returns a three dimensional vector. The Delta prototype that is defined contains three member variables that are initialized with the three supplied arguments. Note the different notations for describing string indices, enclosed in double quotes or being trailed after a dot. The two notations are semantically identical. A method display is also defined that calls the library function print to display the vector in a conventional format on the console. Methods as well as member variables can also be dynamically attached during runtime to objects, allowing great flexibility to the programmer.

Operator overloading in Delta is supported by providing a member function to a table indexed by a string representation of the operator. In the example we overload the operator '+', allowing a natural syntax when adding two vectors in the script.

Sparrow Object

We define a Sparrow object that encapsulates the functionality of the extension interface. Table 3-6 shows the members of the Sparrow object provided to the script programmers. The object encapsulates the supplied functionality of the provided library functions and offers an object-oriented interface to the programmer.

The 'components' member of the sparrow object is an associative table providing access to the registered components of the environment. The programmer can reference any component from this table. Though the table is empty, containing no component handles, it can retrieve them on demand, thanks to the ability to overload the dot operator ('.') in Delta.

components
Member virtual container enabling dynamic access of Sparrow's components to the Delta scripts.

handle	**createcomponent**	(classId[, parent])
Construct and return a handle of a component, optionally specifying a parent component.		
void	**destroycomponent**	({classId | handle})
Destroy a component.		
handle	**getcomponent**	(classId[, serial])
Return a handle to a component.		
StringList	**getcomponents**	(void)
Return a list with the class ids of available components.		
handle	**decorate**	(handle)
Decorate the provided handle by overloading the dot operator enabling dynamic dispatching of component's members.		

Table 3-6 Members of the Sparrow object

Our virtual container approach is similar to the object execution model described in (18). In traditional languages that support static dispatching of object's methods, there is present a method dispatcher for each instance that can dispatch calls to the corresponding method providing it with the call's closure, i.e. the object instance. The method is illustrated in the upper part of Figure 3-1.

Our intention is to mask to the Delta user the details behind function calls to the environment, allowing access of member in a natural way. Dynamic dispatching of member methods allows us to achieve this goal. The dynamic dispatching approach is presented in the lower part of Figure 3-1. The object's method dispatcher this time invokes a single user function supplying it the closure (object instance) and the accessed member and allows the programmer to implement logic that dispatches dynamically the invocation.

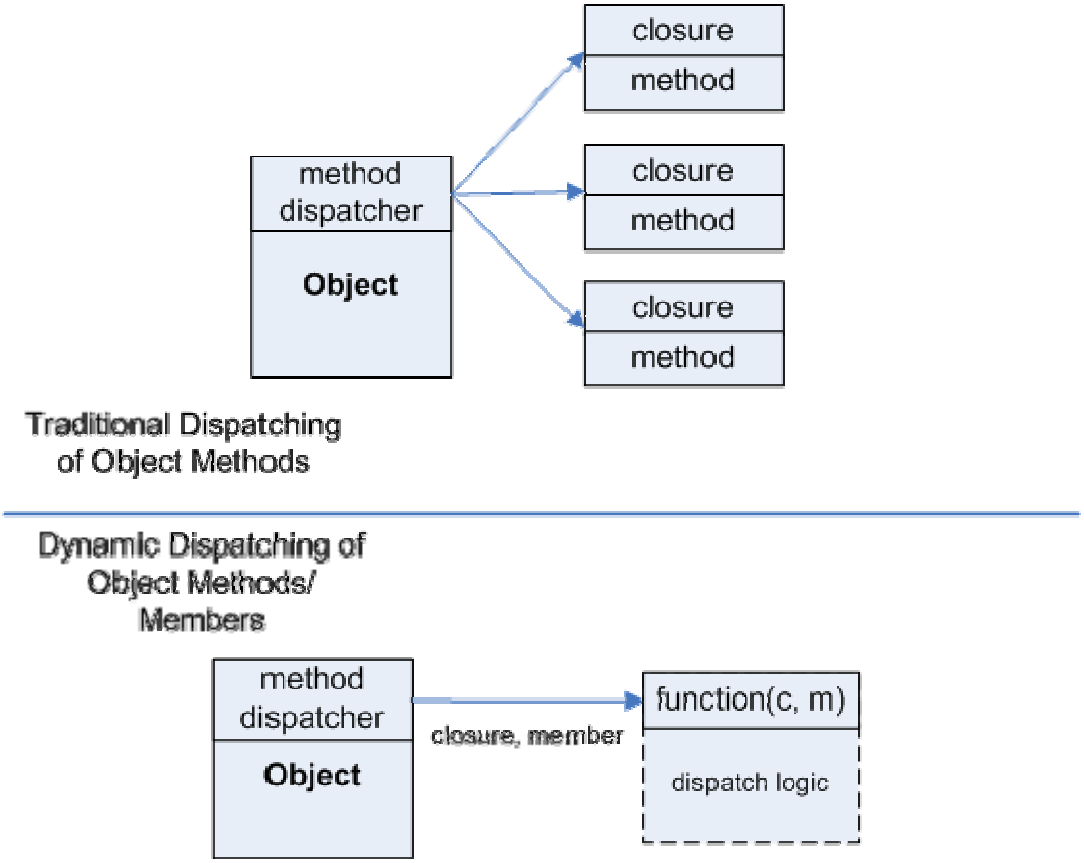

Figure 3-1 Static and dynamic models of method dispathing

In the Delta language we can implement the dynamic dispatching of method call by overloading the dot operator. Listing 3-3 presents the dispatching function implementation of the 'components' virtual container.

```
[ { ".." : (
      function(t, classId) {
            //-- retrieve component entry
            local handle = spw_getcomponent(classId);

            //-- allow the component to load exported
            //   functions on demand
      handle[".."] = component_dispatcher;

            return handle;
      }
)} ]
```

Listing 3-3 Definition of sparrow.components virtual container

When the dot operator in a Delta object is overloaded and the programmer is trying to access a member of that object, either with the dot or the subscription notation, the function attached to the dot operator is called instead. Overloading of the dot operator thus blocks access of members in an associative table. The associated function is invoked with the table itself (closure) as the first argument and the requested member as the second one.

The 'components' object has an overloaded the double dot '..' operator. The double dot operator is similar to the dot operator in Delta, having identical invocation syntax. However they differ in respect to the inheritance handling. If the initial member inquiry is done for an object overloading dot, no inheritance lookup is performed, instead the dot function is called, thus dot ignores inheritance. If in the inheritance lookup chain there is an object overloading dot, then during inheritance lookup this operator is not called, instead the local lookup is always performed, thus inheritance ignores dot.

Local member lookup to an object is made with the following order; (1) the double dot operator, if defined, (2) the dot operator, if defined and only if the object is a top parent, (3) the native table lookup. In other words, double dot operator respects inheritance, while the dot operator is treated as double dot only if it appears in top parents or to objects without inheritance hierarchy.

As a result, in order to allow the Delta programmer to freely inherit the objects provided from the wrapper library we choose to overload the double dot operator that respects the inheritance implications in the member lookup.

In Listing 3-3 the unnamed function retrieves the handle of the focused component instance that has been requested by the Delta programmer. Furthermore, it overloads the double dot operator of the newly created handle object, installing another dispatch

function, before returning it to the caller. This is a dynamic addition of a member to an already constructed object. The installation of a dispatch function to the handle object makes it also an object whose members are dynamically bound, during access-time, much like the 'components' object. Listing 3-4 shows the definition of the dispatch function that we have installed for the overloaded double dot operator of the handle.

```
function component_dispatcher(handle, index) {
      //-- initially check if index is a property
      local property = spw_getproperty(handle, index);
      if (property != nil)
            return property;

      //-- treat the index as a function, binding
      //   spw_call()'s first arguments
      return bind(spw_call, handle, index);
}
```

Listing 3-4 Definition of component_dispatcher

Function `component_dispatcher` is in the global scope and its first argument is the handle object, whose dot operator we have overloaded (the closure), while the second one is the name of the accessed component member.

The function that we have installed to the handle allows us to access dynamically the properties and the exported functions of a component instance. When a member of handle is accessed we look through its properties for retrieval. If it is not found in its properties then we consider the member a function and return a functor object that binds the first two arguments of `spw_call` library function, the component handle and the function name.

Functors are callable objects in the programming languages. In Delta objects are callable when they overload the operator '`()`'. The generic meta-function bind presented in Listing 3-5 taking as its first argument an arbitrary function f, binds its rest N arguments as the first N arguments of f.

In its definition, bind seems to accept only one argument, the function f. However in Delta a function can be called with a more actual arguments that its formals. The actual arguments can be access at runtime using the arguments keyword, which resolves to a numerically indexed table, containing all the actual arguments with which the function has been called. Essentially the formal arguments are a shortcut for referencing the actuals. The bind function constructs and returns a binder functor, storing the f and its N first arguments. Consequently, function f, with dimension M, is transformed to a function f', with dimension M-N.

```
function bind(f) {
      binder = [ { .f : f }, { .args : [] },
            { "()" : (
                  method() {
                        return self.f(|self.args|, |arguments|);[2]
                  }
            )}
      ];

      for (local i=1; i < tablength(arguments); ++i)
            binder.args[i-1] = arguments[i];
      return binder;
}
```

Listing 3-5 Definition of the generic bind meta-function

Function's `component_dispatcher` final implementation is slightly modified, taking advantage of caching for bounded function calls, improving efficiency. However the details are omitted, in the provided listing, for simplicity and clarity of the code.

The overloading of the dot operator allows the programmer to access a component in a very natural way. Delta allows us to hide the details, to the programmer, of how we retrieve the component handle, creating the illusion that there is a container in the Sparrow object that holds every available component. Correspondingly the access of component members is simplified by the overloading of the dot operator in the handle

[2] The |table_name| notation in Delta denotes the expansion of a table's members to an argument list.

object. Listing 3-6 demonstrates what we have accomplished by providing a Delta library to the client programmer.

```
shell = sparrow.components.Shell;
property = shell.init_profile;

sparrow.components.ProjectManager.OpenWorkspace(
      "D:/My Workspaces/profile_confs.wsp"
);
spw_call(
      “ProjectManager”, “OpenWorkspace”,
      "D:/My Workspaces/profile_confs.wsp"
);
```

Listing 3-6 Example of Delta sparrow library usage

In the example program we retrieve the Shell handle from the `sparrow.components` virtual container. The handle has overloaded the operator dot, so we can retrieve Shell’s property “init_profile” by accessing it, as if it was a member of the handle. The next statement accesses the project manager of the environment calling its `OpenWorkspace()` exported function. We then can call the functor object that is returned by the handle’s overloaded dot operator, passing as an additional argument for the target function, the location of the workspace. The last statement is equivalent to the previous one, though we use directly the library functions provided to Delta by Sparrow. The first approach is evidently more natural to write for the programmer than the second one.

3.3 Controlling Existing Components in Delta

In this section we will discuss how scripts can manipulate the environment and will present some examples of scripts that exploit control of component to configure the environment.

Listing 3-7 shows a script that controls the environment using the Sparrow library. Initially we retrieve the main virtual machine of the environment, using the `vmget` library function. The `vmcall` library function is used to call retrieve the Sparrow object from the main virtual machine. The script carries out two jobs; it restores the last opened workspace and it displays the total lines of code in it. Both operations are provided by components written in Delta. The implementation of those components will be discussed in the next section. However, in this example we can see how they can be controlled.

```
main = vmget("main");
sparrow = vmcall(main, "Sparrow");

//-- open by default last used workspace
restore_workspace = sparrow.createcomponent("restore_workspace");
sparrow.components.ProjectManager.OpenWorkspace(
      restore_workspace.GetURI());

//-- count LOCs in workspace
print_server = sparrow.createcomponent("print_server");
line_counter = sparrow.createcomponent("line_counter", "Shell");

locs = line_counter.CountLines();
print_server.print("-->" + locs + " LOCs in workspace '" +
      restore_workspace.GetName() + "'\n");
```

Listing 3-7 A Delta script controlling the environment

The first component we create is `restore_workspace`, which is able to restore the most recently opened workspace. The component includes no graphical modules, but we can simply use it by scripting commands. We retrieve last workspace's location by the `GetURI()` function the component exports, which we give as an argument to the `OpenWorkspace()` function of the project manager, restoring the previous user's workspace. The environment's project manager is accessed via the components virtual container of the Sparrow object, as described in section 3.2.4.

For the second operation two components are instantiated, the `print_server` and the `line_counter`. The `print_server` provides a `print()` function that can send a string to the environment's output window. The `line_counter` can iterate through the scripts

of the workspace and count their lines of code. It provides a `CountLines()` function that returns the number of source lines that are displayed by the script using the `print_server` component. The `line_counter` instance is created as child component of the environment's Shell component, as it exports user commands. The shell automatically collect user commands of its children component in order to position them in its main menu bar and tool bars.

3.4 Implementation of Components in Delta

Though the manipulation of components from Delta as seen in the previous sections provides several applications, these mostly resolve around automatic configurations of the environment. Extensibility of the environment comes from the ability to build new components in Delta that along with manipulation of already existing ones extend the tools or create new custom solutions. In this section we will discuss the integration of scripts as component in the system and describe the implementation of some script components to make the circularity concept of our system comprehensible

In order for an entity to be considered a component in our platform it should register an entry describing it to the central registry. Dispatching of messages also poses several requirements in a component. Most noticeably is that every component instance should derive from a native C++ base class, defined in the system. For a Delta script to meet the requirements of components a proxy class in C++ is needed, which will allow it to fulfill the requirements of registration and message dispatching. Figure 3-2 shows how the script proxy fits in the system.

The script proxy is a native C++ class that derives from the Component super-class. It contains the Delta VM that runs a script component inside the environment. It is the intermediate for messages between the message router of the system and the scripts, implementing an internal dispatching mechanism making the correspondence of exported API calls to local Delta function calls.

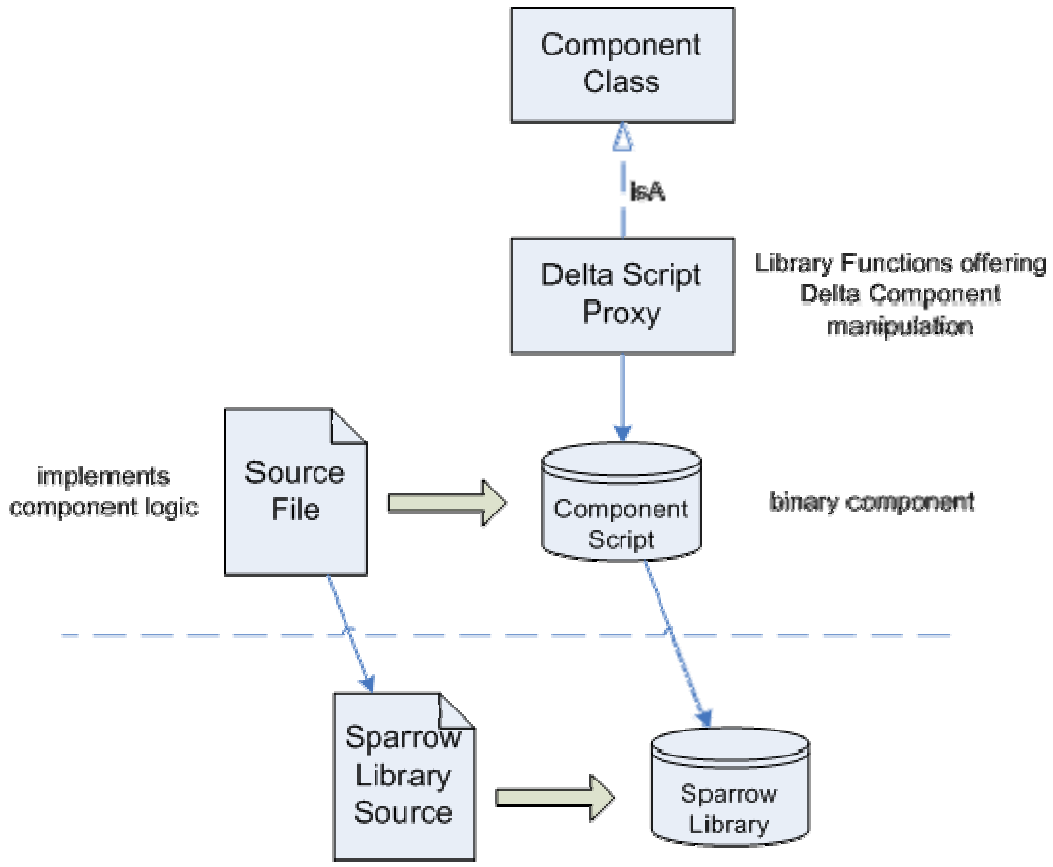

Figure 3-2 Delta script proxy

We will demonstrate how Delta script component are implemented by building two extensions, a workspace restore facility and a line counter, each highlighting different features of the Construction API.

3.4.1 Workspace Restore

Workspace Restore component is a small add-on to the environment written in Delta for bookkeeping of the most recently used workspace from the programmer. In section 3.3 we showed how we can manipulate externally the component to restore the workspace, in an initialization script.

```
function geturi() {
	fp = fileopen("last_workspace", "rb");

	uri = fileread(fp, "string");
	if (uri == nil)
		return "";

	while (not fileend(fp))
		uri += " " + fileread(fp, "string");
	return uri;
}

function onWorkspaceClosed(workspace) {
	if (workspace.serial != 0) {
		uri = workspace.GetURI();
		fp = fileopen("last_workspace", "wb");
		filewrite(fp, uri);
		fileclose(fp);
	}
}

classId = "restore_workspace";
spw_registercomponent(classId);
spw_setcomponentmetadata(
	classId, "Workspace Restorer",
	"A utility for restoring last used workspace",
	"Themistoklis Bourdenas <themis@ics.forth.gr>",
	"alpha"
);

spw_registerfunction(classId, "GetURI", geturi, "String (void)",
	"Returns the uri of the last opened workspace");

spw_registerslot(classId, "onWorkspaceClosed", onWorkspaceClosed,
	"void (Handle workspace)", "WorkspaceClosed");
```

Listing 3-8 Restore Workspace script

In Listing 3-8 the implementation the component is presented. The script initially registers the component and sets its meta-data, giving a user readable name, a short description, its author and version. Finally it registers its functions to the environment along with their signature and documentation. The script registers the local function `geturi()` as the callable object when component's exported function `GetURI()` is invoked. In other words the script supplies names to be used as references to the outer world for invocation of local functions.

A slot is also registered and connected with the `WorkspaceClosed` signal. This signal is emitted when a workspace is closed by the workspace component itself, so that the slot is able to get its location and store it. The two provided function do exactly this, the slot stores the URI of the closed workspace in a file, while the exported function reads the file, retrieving the URI. Signals of already existing components facilitate building of extension on top of them, while they remain unaware of the existence and operation of those extensions.

The script is complete and ready to be loaded in the environment. The procedure clearly illustrates the circularity of the environment, authoring a script inside the environment and using its tools in order to attach it. The script is able to control other components in the environment and extend them, in this case allow the project manager to restore the most recently used workspace.

3.4.2 Workspace Line Counter

The line counter component is another component sample implemented in Delta. It iterates through the workspace's scripts, counts the total lines of code and displays them on the environment's output. It provides two means for accessing its functionality; an exported function that can be invoked by other scripts and a user command it registers to the environment, allowing a user to use it through the user interface.

Listing 3-9 shows part of the implementation of the script component. The code listing demonstrates how a script can register an image in the environment giving it a resource id and use for the display of a user command.

```
main = vmget("main");
spw  = vmcall(main, "Sparrow");
util = vmcall(main, "Utility");

function CountFileLOCs(script) {
      if (typeof(script) != "Table")
            return 0;
      script = spw.getcomponent(script);
      local uri = script.GetURI();
      spw_print("\n", uri);

      fp = fileopen(uri, "rt");
      local lines=0;
      while (not fileend(fp)) {
            filegetline(fp);
            ++lines;
      }
      fileclose(fp);
      spw_print("script: ", uri, " -- LOCs: ", lines, "\n");

      return lines;
}

function CountLOCs() {
      scripts = spw.components.ProjectManager.GetResources("Script");
      util.for_each(scripts, CountFileLOCs);
}

classId = "line_counter";
spw_registercomponent(classId);
spw_setcomponentmetadata(
      classId, "Workspace Line Counter",
      "A utility for counting lines of scripts in opened workspace",
      "Themistoklis Bourdenas <themis@ics.forth.gr>",
      "alpha"
);

spw_registerimage("linecount", "resources/linecount.png");
spw_registercommand(classId, "CountLines", CountLOCs,
      "/{200}Tools/{100}Count Lines", spw.menu_tool,
      "linecount", "Display the total line of codes in the workspace");
```

Listing 3-9 Line Counter script

The script installs a command “Count Lines” under the “Tools” section of the menu bar and the toolbar as indicated in the registration of the command. The command is implemented in the `CountLOCs()` function that retrieves the list of scripts from the project manager of the application, applying on every script a function that counts and displays the lines in the output.

The `for_each()` function that takes as arguments the list of scripts and the `CountFileLOCs()` function is a generic meta-function that iterates through the contents of a list applying to them the supplied function. It is a member of the Utility object that is provided by the main script of Sparrow. Utility offers several generic functions to the programmer and it can be loaded the same way the Sparrow object does, using the `vmcall` library function.

3.5 Interactive Directory for Delta Components

Components in the environment are loaded on demand and the script components are not an exception. There is central component definition directory containing the definition files of all components available to the environment. Components are attached to the system dynamically as they are requested. Currently three kinds of components are supported in the environment:

- build-in components
- components loaded as dynamic libraries (DLLs)
- Delta scripts

The build-in components are basic components of the environment that come with the core of the platform and are necessary for the operation of the system. An example of such a component is the Shell, which is the top-level graphical container of the environment.

Dynamic libraries are native C++ components packed in libraries that can be loaded at runtime. Analogous is the situation with the Delta scripts as discussed in the previous section. The type of each component and its location is described by a definition file that is in the responsibility of the author to write and accompany with each component. We offer an interactive Directory for the Delta Components that aids the component author manage her scripts.

More specifically the facility takes care of:

- authoring of a definition file for Delta components
- compiling and storing the script binary
- automatic recompilations when sources are updated
- removal of components from the system
- interactive loading/unloading of a component

In Figure 3-3 a screenshot of the interactive directory is presented. It illustrates a list of the registered Delta components and their state, whether they are active or not. On top of the component list there is the toolbar with the functionality of the component.

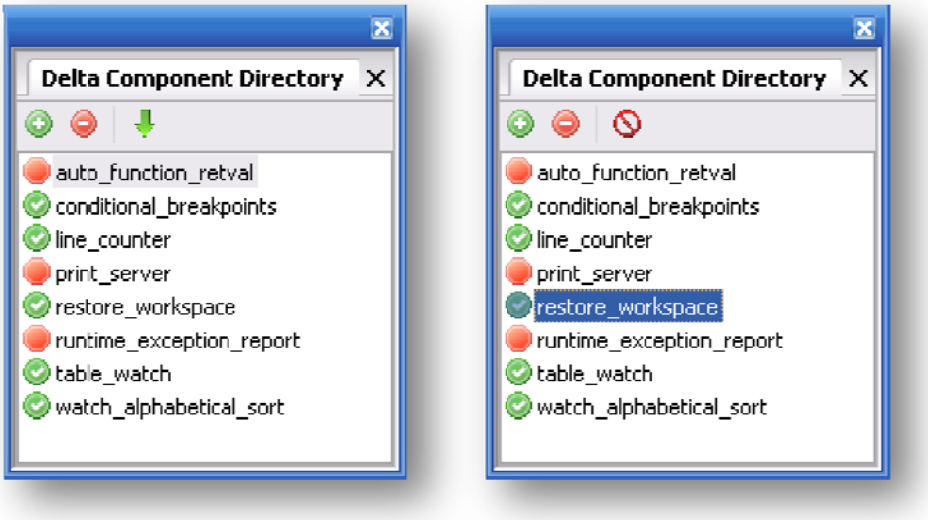

Figure 3-3 Delta Component Directory

The first option is the registration of a new script as a component of the environment. The user will be prompted to select a Delta script to be registered. The selection of script will result to automatically compile the selected script, copy the produced binary file to the location where Delta components are stored. Finally the component definition file will be authored and stored in the appropriate directory.

The second option removes a component from the directory, unregistering it from the system. It also deletes its definition file and the binary code produced.

Finally, the last option allows dynamic loading or unloading of the component from the environment providing a graphical approach for choosing what features will be enabled to the environment, even by a non-advanced user, without requiring authoring of a configuration file.

3.6 Delta Component Directory Extension APIs

Table 3-7 provides the exported API of the graphical directory tool for the Delta scripts. It exports every action already available to the end-user as well as a list of all registered components.

void	**RegisterComponent**	(String uri)
Register a script component to the system by supplying its source URI.		
void	**UnregisterComponent**	(String classId)
Unregister a component from the system.		
Handle	**LoadComponent**	(String classId)
Load a script component in the environment and create an instance, returning a handle to it.		
void	**UnloadComponent**	(String classId)
Unload a component from the environment.		
StringList	**GetAvailableComponents**	(void)
Return a class id list of available Delta script components.		

Table 3-7 Delta Component Directory exported API

4. Source Level Debugger

4.1 Technical Approach

Sparrow's debugger kit for Delta has been implemented as a set of components in the system. The kit is comprised of the debugging extensions to the DeltaVM component, as well as the graphical elements in the environment that support the debugging process. The graphical elements provided are a list view of Breakpoints, the call stack and watches of expressions.

A Delta program can be debugged either in console mode, as a different process, or as a Sparrow script. When debugging as a console program the debugger is running in another console process, controlled by the environment. For their communication, the two processes use TCP/IP sockets. The debugger is the server waiting control messages from the debugging client, the IDE.

If a program is debugged as a Sparrow script, it is implied that the script will require the facilities provided by the Sparrow environment (library functions, component registry, etc.). In this case a new instance of the environment is spawned as a separate process and the script is running inside the new environment. We avoid running the debugged script inside the working environment as this could affect the stability and its behavior. The script can control every component in the environment, doing so while the programmer is trying to debug it can produce confusion and frustration. Thus we choose to apply the script in a new program instance. Again the communication between the two processes happens over TCP/IP sockets, similarly to the console applications.

4.2 Delta Debugger Support

The debugger functionality has been incorporated in the DeltaVM component, which in general controls the execution of every script in the environment. Figure 4-1 illustrates the way the Delta language is incorporated in Sparrow IDE. Details of internal Sparrow IDE components that are interfacing with the actual delta building blocks are omitted for simplicity in the figure, as they are described in later sections.

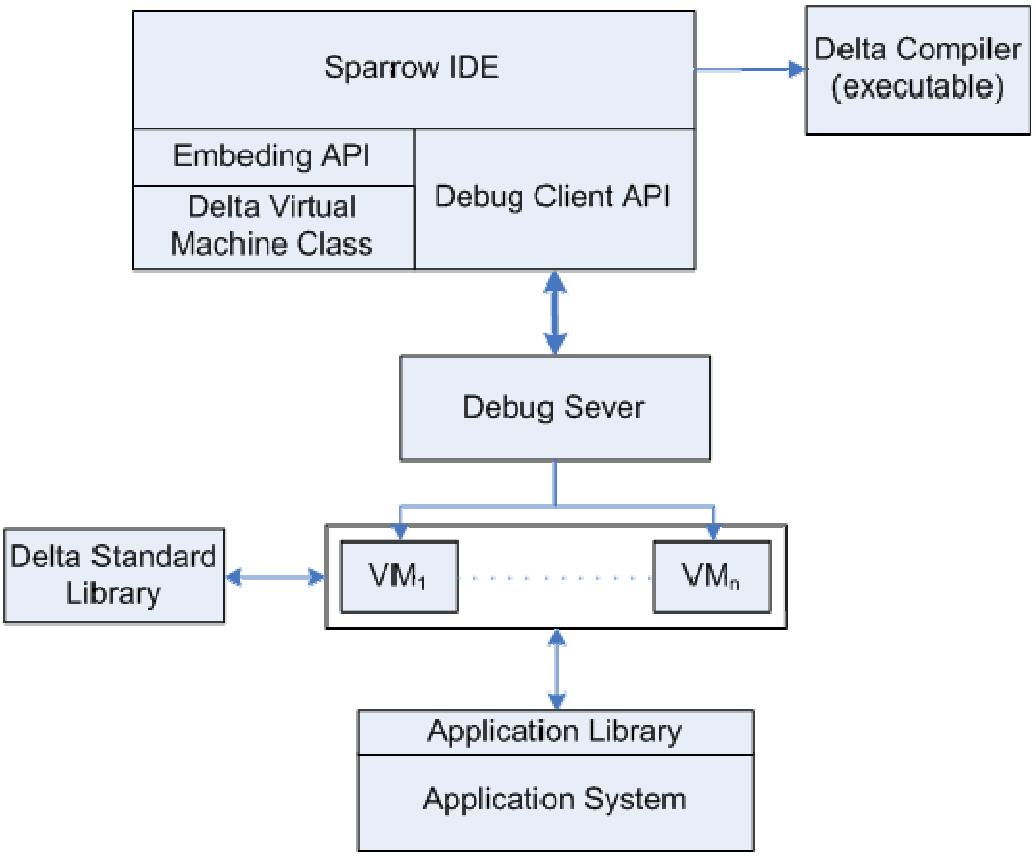

Figure 4-1 Delta debugger incorporation in Sparrow IDE

The Delta compiler executable is used as an external process invoked by the system for the building process of projects. The Delta virtual machine that is running inside the environment is interfacing using the provided embedding API. For the debugging of Delta programs the Delta debug server is used. Sparrow IDE implements the

functionality of a Delta debug client that connects to the server to debug the script in a virtual machine. The debug client communicates with the server over a lightweight TCP/IP protocol. The debug server holds the required virtual machines for the program to run. Each Delta script runs in its own virtual machine. The virtual machines can access the Delta standard library and other application libraries that scripts use. For instance the application library could be the extension API that Sparrow IDE offers to Delta, and the Application system, Sparrow IDE itself.

The functions facilitating the debugging of programs, a subset of DeltaVM's exported API, are presented in Table 4-2. The API allows insertion of breakpoints and manipulation of the execution of a program. The debug actions are generic enough to meet the requirements of a large set of language, as they describe high level concepts of the debugging process instead of being too specific and constrained in the language. When the execution halts in a breakpoint operations on the call stack and evaluation of expressions can be processed, aiding facilities for expression watches.

The communication between the client and the server happens asynchronously thus the client has a dedicated thread listening for responses from the debugger including the halt on a breakpoints, runtime failures or graceful stop of execution. DeltaVM component also manages the breakpoint holder, containing the breakpoints placed by the user for the whole workspace. These breakpoints are sent synchronously at the startup of the connection between the IDE and the debugger. Breakpoints that are added during the debugging process are propagated asynchronously to the debug server.

Finally DeltaVM triggers signals concerning a debugging session, such as halting on a breakpoint, the initiation or the end of a debugging session. These signals are processed by other components in the environment that express interest in those events, building the debug system in Sparrow.

In the following sections we will describe the major debugging facilities of our environment.

4.3 Breakpoints

The breakpoints are marks in the source code that indicate halt of execution during a debugging session. They are identified by a source file and a line of code, so as to when the execution reaches that line, it temporarily stops, in order for the programmer to assess the state of the program variables. The DeltaVM component contains the central breakpoint holder, aggregating every breakpoint in a workspace and propagating them to the debug server when a debugging session is initiated.

Conditional breakpoints are supported by the environment. They are handled similar to the rest breakpoints though they carry a condition expression that must be satisfied for the execution to halt when the breakpoint is hit. The evaluation of expression occurs every time the breakpoint is hit by the debug server.

Lines with breakpoints attached are marked in the source editor for the programmer to review and navigate during the debug process. It is possible for the programmer to place a breakpoint to an invalid line. Invalid are considered the lines that contain no executable code; for instance an empty line or a function declaration. When invalid breakpoints are transmitted to the debug server, it tries to resolve them by moving the break to the next available line, if possible. Those corrections are propagated back to the debug client and the breakpoints are fixed automatically in the source editor.

In addition to the source editor marking, a graphical tool presenting a list view of a workspace's breakpoints has been constructed. The list of breakpoints is presented in Figure 4-2, at the bottom of the window. It aggregates the breakpoints of a workspace indicating their source file, line and condition. It also provides a counter of the hits of a

breakpoint during the last debugging session. Apart from providing a quick view of the breakpoints placed, the tool also permits quick navigation to the source code of the breakpoints by selecting an item from the list. Most importantly it allows editing of breakpoint's hit condition expression. The condition is edited using the dialog shown in the screenshot.

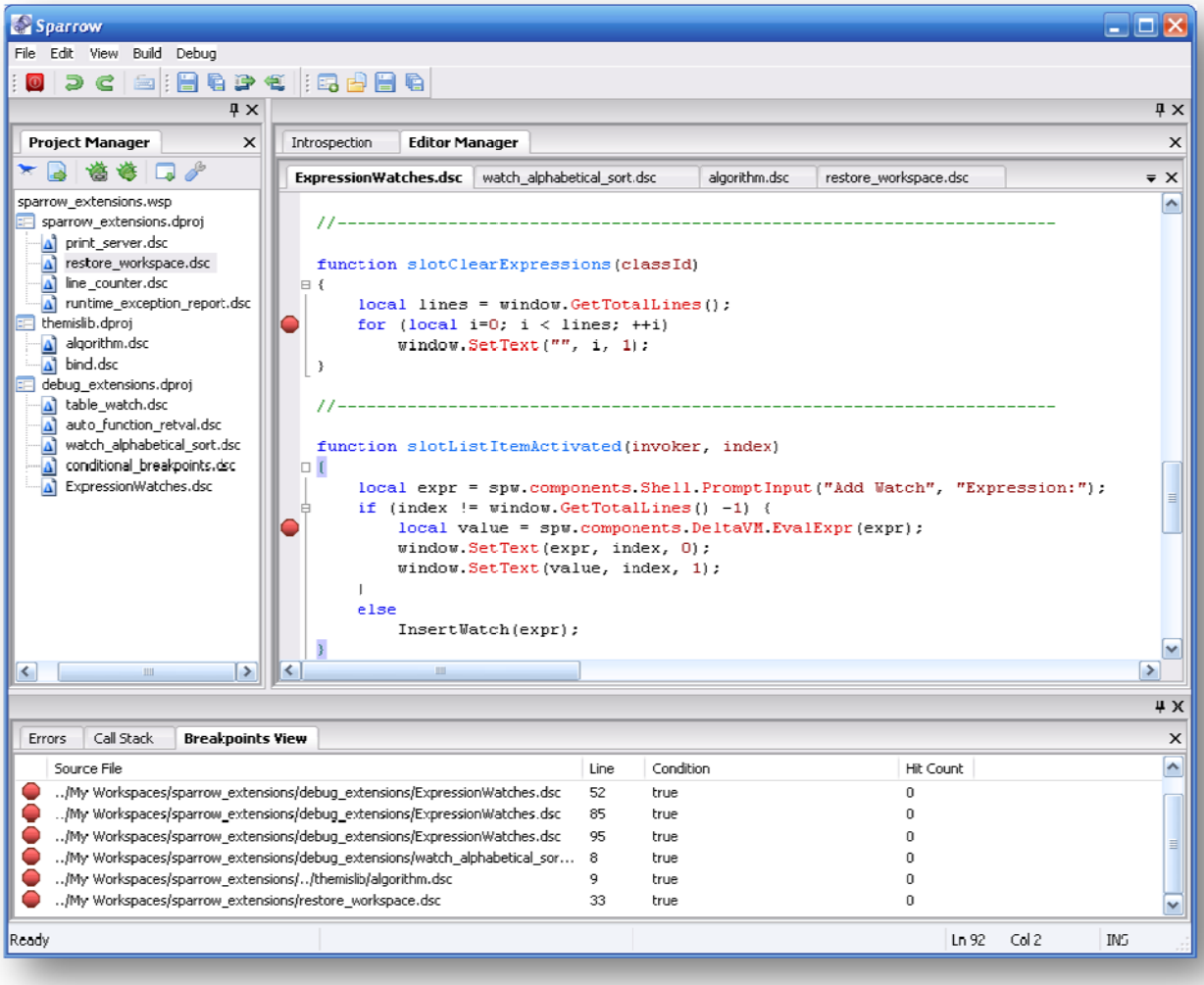

Figure 4-2 Breakpoints in the source editor and the list view

The component is entirely implemented in Delta. Delta script components are able to render graphical interfaces as they lack a GUI toolkit library. However Sparrow offers a

core component, the ListViewComponent, which offers an API for creation of a multi-column list view in the environment. BreakpointsView component instantiate the ListViewComponent and manages its content presenting the breakpoints of the workspace.

4.4 Call Stack

The call stack during the debugging of a program presents the function activation records and their order of invocation, giving a view of the programs execution. Debuggers usually offer a view of the program's call stack of a program at breakpoints halts. Programmers can navigate on the stack, inspecting all the activation records in the stack, changing the scope of the watched variables.

Sparrow provides an interactive graphical tool for call stack inspection. Figure 4-3 shows a snapshot of the tool. The information contained in the window is focused on the activation record of a function. The name of a function is presented, its definition and invocation line, the scope level of the function, and its arguments during the call. Additionally the definition source file is presented. In Delta every script runs inside its own virtual machine. It is possible for a script to load another one, loading it to another VM to execute. The former script has access to the latter's global functions, being able to call them. As a result a stack view may contain functions residing in different virtual machines. All functions are aggregated under a common stack and for each one its owner virtual machine is indicated in the stack view.

The tool is interactive, so the user can navigate through the call stack to inspect the context for each function call, by selecting a stack frame in the window. Moving the stack brings in effect the context of each activation record thus allows inspection of its local variables. As is the case with the BreakpointsView this tool is also written in Delta, using the ListViewComponent.

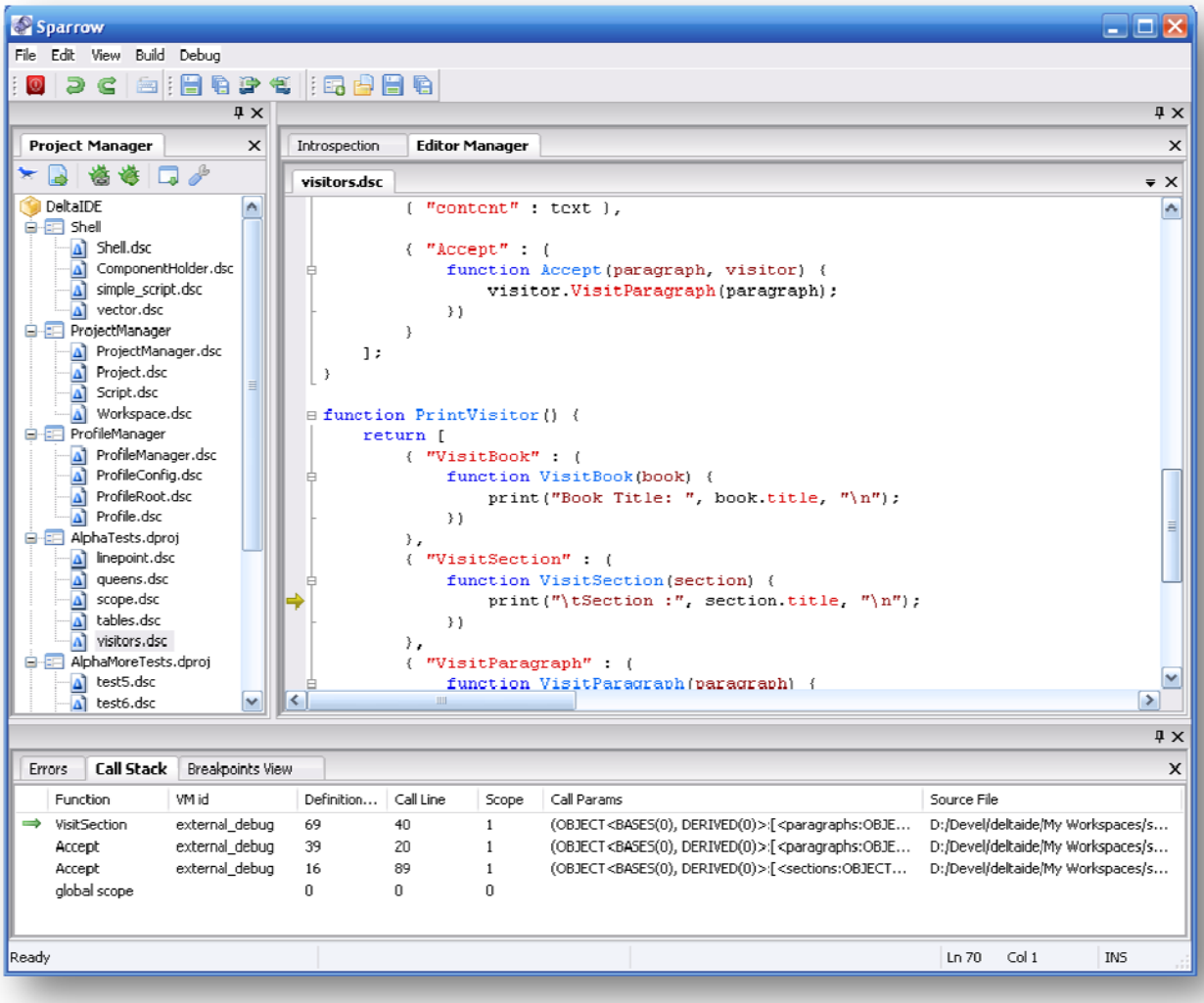

Figure 4-3 Call Stack window during debugging

In Table 4-4 the exported API of the tool is shown. Navigation on the stack and retrieval of activation records are the provided functionality. Though the API is generic enough for the tool to be considered language independent this is not the case, as the display is closely coupled with Delta including virtual machine identifiers and function scope level, information that is not compatible with other languages like C++ or Python.

4.5 Watches

During the breakpoints programmers need to examine the state of the program's context, such as the local variables of a function, as well as the state of their objects. Watches are used to monitor variables during debugging of a program and are re-evaluated every time a breakpoint is hit or the variables are invalidated due to movement of the stack pointer.

A component has been implemented for Sparrow that provides a user interface to the programmer to insert and manage her expression watches. A snapshot of application is illustrated in Figure 4-4, where the watches pane is on the bottom of the window. The arrow symbol in the source editor indicates the current position of the debug execution. The watched expressions are evaluated at the indicated scope of the program. User can insert or modify her watches using the "Add Watch" dialog, floating in the picture, which appears when selecting a watch in the pane.

The watches are written entirely in Delta as a script component inside the Sparrow on top of the ListViewComponent, which it instantiates and controls. Similar to the BreakpointsView component it is based on the basic graphical facilities the ListViewComponent offers. It then manipulates the control to provide a list of expression watches in the system.

Delta debugger in addition to variable watches supports watches on expressions that follow syntax similar to Delta, with a few additions to provide scope resolution. The expression evaluator though has some limitation, excluding some valid Delta expressions, such as function call, as they require code execution that has side-effects.

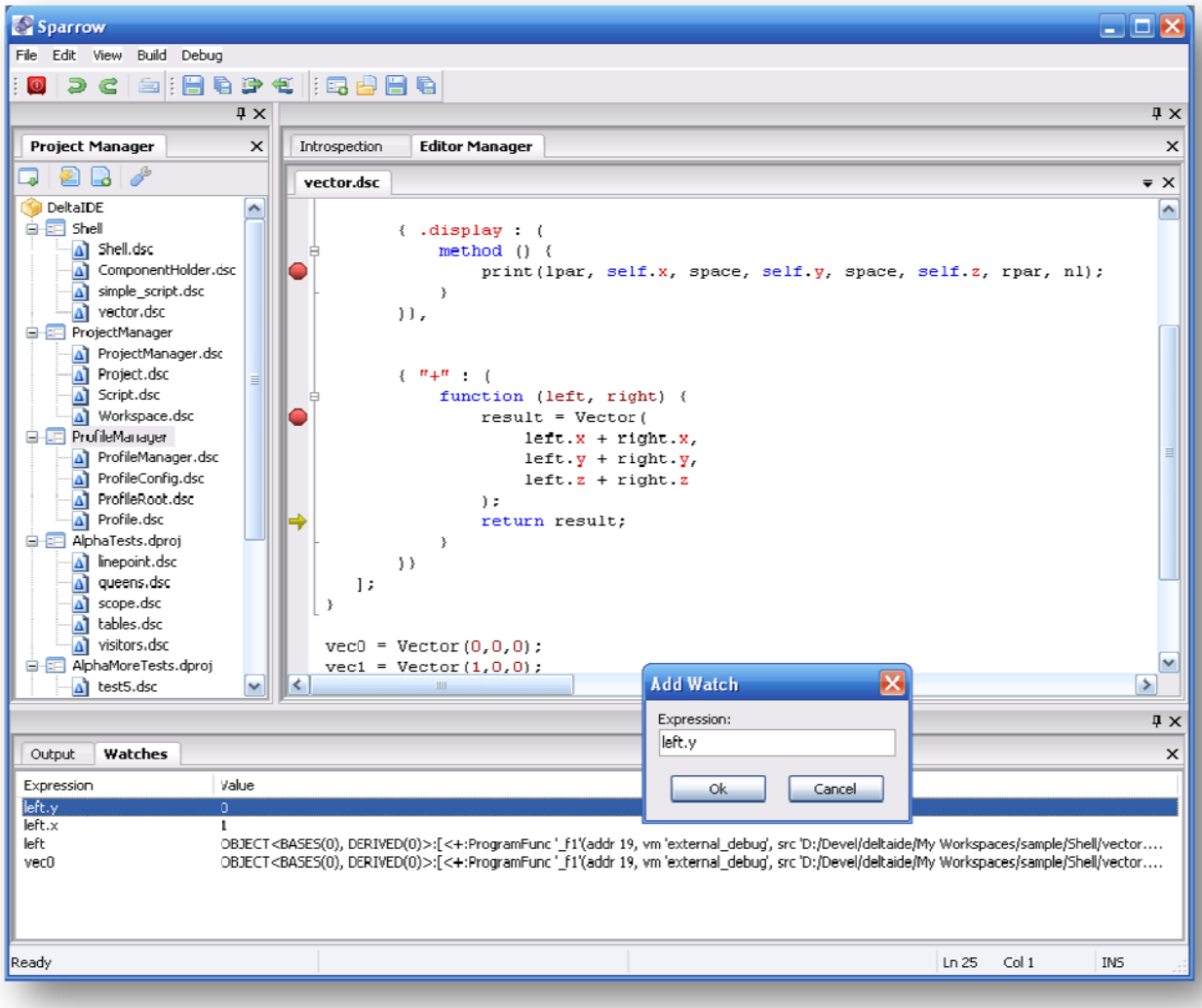

Figure 4-4 Debug Watch Window

The component further allows modification of a variable's value by the user. The new value is propagated to the debugger, updating the variable in the virtual machine's memory. Consequently, the programmer is allowed to make changes on-the-fly and experiment with alternatives during debugging.

Like in most graphical components of the environment, a control API is exported that allows extensions on top of the basic watch window. A programmer can construct

specialized watch windows that better suit her requirements during debugging. Some extension examples are demonstrated in the next section.

4.6 Delta Debugging Extensions

In section 3.4 we discussed how we can implement components in Delta that extend the environment. In this section we will briefly discuss two extensions on the debugging facilities, implemented in Delta.

4.6.1 Function Return Value Auto Watch

Apart from watches placed by the user, there are some automatic values that are sent directly from the debug server whenever a breakpoint is hit. These are the results of functions called just before the halt of execution due to the breakpoint.

These values are not displayed in the watches window described in section 4.5 as it explicitly displays user watches. Another Delta extension has been implemented using the ListViewComponent of the environment that aggregates all these messages from the debug server and presents them to the user.

Similarly to the ExpressionWatches, the extension registers slots to the DeltaVM component's signals for capturing the messages of function return values from the debugger and then it automatically populates the list. The information is persistent until the scope context of the activation record changes in which case the list is reset.

4.6.2 Delta Object Watch Decorator

As seen in Figure 4-4 the display of an object's content in the watch window is rather complex, revealing a lot of implementation details not helpful to the user, adding clutter to the watch window. The watch window displays the objects exactly as it receives it

from the debugger. An extension has been implemented over the ExpressionWatches that decorates the display of objects, presenting a more helpful, human readable representation.

During the debugging, users usually are more focused on the member variables of an object instead of its member functions. Thus the member variables and their values are presented in front of the functions. Further to pushing the display of functions behind we remove most of the clutter keeping only the name information of the member function.

Arguably, when compared to the default, this is a better presentation of an object, which can easily be applied or removed by the environment's user. Most notably, the purpose of the extension is to demonstrate the potential given to the user to custom-tailor her environment to her preferences.

4.7 Debugger Components Extension APIs

Table 4-1 presents the signals fired by the virtual machine module of the environment. The debugging facilities presented in previous sections are based on connecting slots to those signals. The signal mechanism allows extension of the build and debugging systems while maintaining loose dependencies among concerned components.

void	**sigBreakpointAdded**	(String uri, uint line, String condition)
Signal fired by insertion of a breakpoint.		
void	**sigBreakpointRemoved**	(String uri, uint line)
Signal fired by removal of a breakpoint.		
void	**sigBreakpointHit**	(String uri, uint line)
Signal fired by halt of debugging to an active breakpoint.		
void	**sigPushStackFrame**	(String record, uint defLine, uint callLine, uint scope, String params)
Signal fired by insertion of a new activation record in the call stack.		
void	**sigInvalidateWatches**	(void)
Signal fired when expression watches become invalid.		
void	**sigDebugStarted**	(String uri)
Signal fired when debug process is initiated.		
void	**sigDebugStopped**	(String uri)
Signal fired when debug process is finished.		
void	**sigDebugResumed**	(void)
Signal fired when debug process is resumed.		
void	**sigRunStarted**	(String uri)
Signal fired when execution of a script is initiated.		
void	**sigRunStopped**	(String uri)
Signal fired when execution of a script is finished.		

Table 4-1 DeltaVM signals

In Table 4-2 the exported API of the DeltaVM is provided. It defines the concept of the runtime and debugger of a language.

void **Run** (String uri)
Execute the specified script as a Sparrow script, in the same process.

void **RunConsole** (String uri)
Execute the specified script as console program, in a separate process.

void **Debug** (String uri)
Start debugging the specified script as a Sparrow script, in a new instance of the environment.

void **DebugConsole** (String uri)
Start debugging the specified script as console program, in a separate process.

void **StopDebug** (void)
Stop debugging of the currently running script.

void **ToggleBreakpoint** (String uri, int line, String condition)
Toggle a breakpoint at the specified location. Returns true if breakpoint is set, false otherwise.

void **ClearBreakpoints** (void)
Clear all breakpoints placed in the script.

String **EvalExpr** (String expr)
Evaluate an expression returning its value in a string format.

void **StepOver** (void)
Make a step in the debugging, without entering in any function call.

void **StepIn** (void)
Make a step in the debugging, visiting function calls.

void **StepOut** (void)
Make a step in the debugging, exiting the current function.

void **RunToCursor** (void)
Run the program in debug mode, until the line of the cursor is reached.

uint **GetTotalStackFrames** (void)
Return total number of stack frames.

uint **GetCurrentStackFrameIndex** (void)
Return index of the current stack frame.

String **GetStackFrame** (uint index)
Return information description of the specified stack frame.

void **MoveStackFrame** (uint index)
Move the stack frame to the specified index.

Table 4-2 DeltaVM exported API

In Table 4-3 the API exported by the BreakpointsView component can be found. The control does not allow insertion or removal of breakpoints as the list is merely a view of the available breakpoints. Insertion and removal is handled by the DeltaVM component, which wraps the Delta debugger. However, the API supports addition of conditions on available breakpoints as the control itself provides an interactive dialog for the user to modify breakpoints' conditions.

String	**GetBreakpoint**	(uint index)
Return the breakpoint in the specified index.		
String	**GetCondition**	(uint index)
Return the condition of the specified breakpoint.		
void	**SetCondition**	(uint index, String condition)
Set a condition for the specified breakpoint.		
uint	**GetTotalBreakpoints**	(void)
Return number of total breakpoints.		
void	**Clear**	(void)
Clear all breakpoints.		
bool	**slotInsertBreakpoint**	(String classId, String uri, uint line, String condition)
Slot for sigBreakpointAdded signal.		
void	**slotRemoveBreakpoint**	(String classId, String uri, uint line)
Slot for sigBreakpointRemoved signal.		

Table 4-3 BreakpointsView exported API and slots

Table 4-4 shows the exported API of the CallStackView component. It allows inspection and manipulation of the call stack at a breakpoint. Also it connects slots to the debugger's signals to automatically update the call stack. It lacks any function of insertion or removal of the stack frames to prohibit scripts from modifying it, as stack information is strictly provided only from the debugger using signals.

uint	**GetTotalStackFrames**	(void)
Return total number of stack frames in the call stack.		
String	**GetStackFrame**	(uint index)
Return stack frame with the specified index.		
void	**MoveStackFrame**	(uint index)
Move stack pointer to specified index.		
void	**slotPushStackFrame**	(String classId, String record, uint defLine, uint callLine, uint scope, String params)
Slot for sigPushStackFrame signal.		
void	**slotClear**	(void)
Slot for sigDebugResume signal.		

Table 4-4 CallStackView exported API and slots

Table 4-5 lists the API of the ExpressionWatches component. The component is controlling the ListViewComponent for provision of the watches window. It connects its slots to the debugger's signals for invalidation of components and uses the DeltaVM component's interface for evaluation of watched expressions.

void	**InsertWatch**	(String expression)
Insert a watch for an expression.		
void	**RemoveWatch**	(int index)
Remove the watch with specified index.		
String	**GetWatch**	(int index)
Return watched expression with specified index.		
void	**Clear**	(void)
Clear the watches window.		
void	**slotUpdateWatches**	(String classId)
Slot for sigInvalidateWatches signal.		
void	**slotClearExpressions**	(String classId)
Slot for sigDebugStopped signal.		
void	**slotListItemActivated**	(Handle invoker, uint index)
Slot for sigListItemActivated signal.		

Table 4-5 ExpressionWatches exported API and slots

Table 4-6 presents the API of the ListViewComponent, a component provided by Sparrow's core for displaying collections in a multi-column list view.

void **SetTitle** (String title)
Set the window's title.

void **SetColumns** (StringList columns)
Set the column titles of the list view.

void **SetImages** (StringList images)
Set image ids that will be used by the list view.

void **Append** (String line)
Append a line at the end of the list; fields are separated by '#' delimiter.

void **Insert** (String line, uint index)
Insert a line in the list; fields are separated by '#' delimiter.

void **Remove** (uint index)
Remove a line from the list.

uint **GetTotalLines** (void)
Return number of total lines in the list.

String **GetText** (uint row, uint col)
Return text in the specified field of the list view.

void **SetText** (String text, uint row, uint col)
Set text in the specified field of the list view.

String **GetImage** (uint row, uint col)
Return image id of the specified field of the list view.

void **SetImage** (String image, uint row, uint col)
Set image id in the specified field of the list view.

void **Clear** (void)
Clear all elements of the list.

void **sigListItemActivated** (uint index)
Signal fired by activation of an element in the list.

void **sigListItemContext** (uint index)
Signal fired by requesting of a context menu for an element in the list.

Table 4-6 ListViewComponent exported API and signals

5. Runtime Configuration and Adaptation Infrastructure

5.1 Technical Approach

Dynamic substitution of components in the environment entails that the application's modules that are not statically bound together. Components are interchangeable as long as they follow a certain policy for communication with their outer world. A policy defines an exported API that components should provide to be compatible for interoperation among them. Furthermore dynamic substitution implies that it is possible to supply some alternative implementations for parts of the system and choose among them which one should be instantiated based on decisions made at bootstrapping. This attribute is described as adaptation of the environment to the user's needs.

Sparrow is a good candidate for an adaptable environment due to its modular architecture, which allows dynamic assembly of environment's parts at runtime. Modularity and dynamic assembly of components is a partial requirement for characterizing a system adaptable. In addition, an infrastructure for describing the actual requirements and constraints of the users remains to be addressed, in order to exploit the ability of dynamic synthesis.

Configuration and adaptation in the environment resolves around the concept of user profiles. Profiles contain user constraints or instructions for building the environment and define its layout. Profiles are centrally managed by an adaptation manager component in Sparrow. The means for describing those requirements is left open by the system, accepting extensions for refining the adaptations. Two complementary methods have been incorporated thus far; configuration files that describe user attribute-value pairs and Delta scripting, which can construct the environment's layout by manipulating its main components.

Attribute based configuration files are simplistic and their expressive power is limited. They are usually used to describe attributes of components, such as the fonts of the editor, the color or their size. We would prefer to have the adaptations describe user's properties instead. The user could provide her profile there and then the environment would adapt to that profile based on decisions dictated by user's characteristics.

This kind of approach is adaptation before use, based on rules to extract decisions accepting as input a user profile. Rules in Sparrow are described using a domain specific language, DMSL. Decision Making Specification Language is a mechanism that provided a rule file and a user profile it can extract decisions on the activation of components in the system.

5.1.1 DMSL Overview

The output of the decision making process of DMSL (7) is a set of activation and deactivation commands. An activation translates to inclusion of a component during the dynamic synthesis process while a deactivation its exclusion from the final assembly. The decision logic is described in a form of if..else statements in independent decision blocks. DMSL was originally created for adaptation of visual component; however it is general enough to be easily transferred to other application fields.

Listing 5-1 shows a sample program in DMSL, which could describe the decision logic for the target language in Sparrow. The program takes the decision on which language should be activated, according to the preferred language in user's profile. The decision logic takes in consideration two components of the environment; the language compiler and the runtime environment. If the language of preference is Delta the decision logic activates the DeltaCompiler as the system's compiler and the DeltaVM as its runtime environment. Alternatively if Python was the language of preference the

PythonInterpreter component is activated as both the compiler and runtime, as Python is an interpreted language.

```
stereotype Delta : params.user.language = delta
stereotype Python : params.user.language = python

component Compiler [
      if Delta then
            activate DeltaCompiler
      else if Python then
            activate PythonInterpreter
]

component Runtime [
      if Delta then
            activate DeltaVM
      else if Python then
            activate PythonInterpreter
]
```

Listing 5-1 Language activation program in DMSL

Stereotypes are a construct of language for creating aliases for boolean expressions. DMSL supports forming of hierarchies for the configuration attributes, which are essentially dot separated identifiers. The keyword `params` is the root of a configuration hierarchy, under which every attribute resides. In the example we use the language attribute of the user to access her language of preference.

We should make clear that a component block in DMSL does not necessarily correspond to a Sparrow component. The component keyword in DMSL indicates the beginning of decision block definition. Sparrow components can request the evaluation of any component block of DMSL, retrieve its activation decisions and configure itself by applying the activation results in its own context.

5.2 Shell

Before we discuss how the configuration of the environment happens in Sparrow, via scripts and DMSL rule files, we should briefly describe the Shell, a core component of the environment. Shell is the top level container of all visual components, the main frame of the application providing docking areas for graphical components to be deployed.

Table 5-1 presents Shell's API. It provides functions for inserting and removing graphical components to its docking areas allowing other components, but most importantly scripts, to deploy components in the environment or move them around.

The Construct function accepts as an argument the identifier of a profile for construction of the environment. Shell can retrieve the specified profile from the adaptation manager, where profiles are stored, to apply it. The execution of a profile has as a result the destruction of the current Shell layout and its reconstruction based on the profile's instructions. The Shell stores as a property value the name of the profile that should be activated at start-up in order to build itself. However the user may choose a new profile during runtime hot-swapping between profiles.

Shell component also manages the main menu bar and tool bars of the application. Both bars host user command defined by components. The shell collects exported user commands from its immediate children components and displays them to positions defined by their supplied paths. The commands are removed automatically when a component is removed from the shell as it can no longer serve its commands. In addition there is a set of exported functions that allows components that are not immediate children of the shell to supply commands to the application's menus. The functions accept a command description, containing a component callback and potentially an image, associated with user command's path. The shell also allows

invocation of commands from scripts and component by providing the ExecuteUserCommand function.

5.3 Adaptation Manager

Adaptation manager is a component of the system, which is the central point of profile manipulation and configuration of the environment. As a graphical component, presented in Figure 5-1, it provides a list of the available adaptations in the environment for user to select. The adaptations are stored in the local file system of the application and the adaptation manager is responsible for scanning system directories, locating and loading available profiles. A profile is a container of configuration and adaptation units of the system.

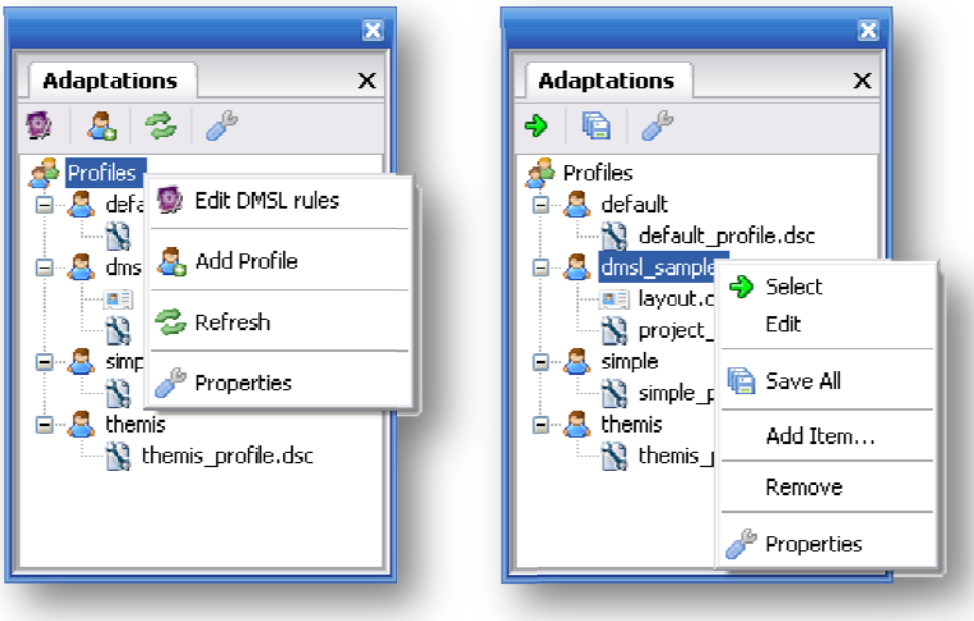

Figure 5-1 Adaptation Manager window

Adaptation Manager being a framework component shares its core design with the Project Manager, discussed in section 6.1. The approach for environment configuration

remains open, providing a framework for adding extension components that offer configuration implementations. Sparrow provides two types of configuration out-of-the-box; profile configurations that describe user's preferences and constraints, and Delta scripts that manipulate the environment.

Profiles can be browsed from the Adaptation Manager, modified and tested on the fly. The user may choose to hot-swap her profile during runtime which would result to rebuilt of the environment. It also provides an interface for creation of new profiles in the application and addition of configuration elements.

Under the hood, the adaptation manager maintains the configuration scripts, monitoring them for modifications and automatically recompiling them to have the most recent version each time they are applied.

Finally it is the point of access of the central DMSL rule file of the system. Our decision to have one central DMSL rule file comes from the fact that the decision logic for adaptation of the system should be centralized and activation of certain parts may result to activation or deactivation of other parts in the system, requiring the logic to take into consideration all activations. Consequently, a component author requiring decision extraction for adaptation of her component should edit the central rule file and describe its adaptation logic. Nonetheless this fact does not constrain the programmer from using an independent rule file, which should be her responsibility to parse, in order to perform localized adaptation of her component.

5.4 DMSL Rules

Listing 5-2 gives an impression of Sparrows adaptability presenting the decision logic of the Shell layout based on user's attributes. The DMSL program starts by describing four categories of users, the Delta beginner and advanced user, which are mutually exclusive characterizations, as well as the debugging programmer and extension author. The idea behind the adaptation logic is to provide to each user the tools that are adequate and necessary for her job.

```
//-- Decision Making logic for Sparrow Shell
stereotype DeltaBeginnerUser : params.user.delta_exp = beginner
stereotype DeltaAdvancedUser : params.user.delta_exp = advanced

stereotype Debug : params.user.debug = true
stereotype ExtensionAuthor : params.user.extension_author = true

component Shell [

      activate ProjectManager
      activate Output

      if hasattr(user.delta_exp) and DeltaBeginnerUser then
            activate WelcomePage

      if hasattr(user.delta_exp) and DeltaAdvancedUser then [
            activate AdaptationManager
            activate ErrorList
      ]

      if hasattr(user.debug) and Debug then [
            activate ExpressionWatches
            activate CallStackView
            activate BreakpointsView
      ]

      if hasattr(user.extension_author) and ExtensionAuthor then [
            activate DeltaComponentDirectory
            activate ComponentSpy
      ]
]
```

Listing 5-2 Sparrow DMSL rules for Shell layout

The program activates unconditionally the project manager and the output window of the environment as they are components that are most certainly used by every user. Then the decision logic begins deciding that if the user is a beginner it should display

him a welcome page that introduces the environment. On the other hand for advanced users of the environment it immediately activates more advanced tools like the adaptation manager that allows manipulation of profiles and the error list, which is an alternative view of the output window, aggregating error messages of the compiler. Those components are hidden from the beginner user to avoid intimidation by a complicated looking interface, permitting him to discover it alone, when she feels more comfortable with the environment.

The debug property of the user denotes the requirement of the user to have instantly access to the debugging facilities of the environment, instead of manually enabling them from the menus. We provide this option again to avoid cluttered windows when need for debugging is not required by a programmer. Finally extension authors require activation of tools that are of no use to regular developer of external applications, thus the Delta component directory, discussed in section 3.5, and the component spy are activated only to those users. The component spy is a visual tool providing introspection of environment's components to extension authors, presenting their properties, exported API and documentation.

```
user.delta_exp = advanced
user.debug = true
user.extension_author = true
```

Listing 5-3 User Configuration

Listing 5-3 shows a user configuration file that would describe her preferences of the environment. The configuration portrays an advanced user that wants debug facilities and is also an extension author. By describing a small set of characteristics the environment can extract a decision that would result to the instantiation of the environment.

When a profile is applied the shell is provided with a configuration similar to the one in Listing 5-3, which in turn supplies to the centralized DMSL unit to process with the

decision logic and retrieve the results, as a series of components to deploy in its docking areas.

5.5 Configuration Scripts

The decision making language that described thus far and is incorporated in Sparrow allows for vast adaptation to the user's needs. However it poses one inherent constraint. DMSL is a rule based language without any side-effects; it can only return decision to the adaptation unit. The actions taken by the parser's decisions can not be affected by the end-user but from the adaptation's unit author. In other words this kind of configuration limits the possible effects on the system to a set of predefined configurations from the programmer. The adaptation logic is embedded internally in the component. As a result components may not be flexible to addition of new extensions.

Configuration scripts are more expressive, allowing the adaptation logic to be described externally in Delta scripts by the end-users. The process of controlling Sparrow from Delta scripts has already been described in section 3.3. In this section we will present an example of such script configuring the environment.

```
main = vmget("main");
sparrow = vmcall(main, "Sparrow");

//-- configuring the layout of the environment
sparrow.components.Shell.AddComponent("ProjectManager",
      sparrow.upper_left);
sparrow.components.Shell.AddComponent("AdaptationManager",
      sparrow.upper_right);
sparrow.components.Shell.AddComponent("Output",
      sparrow.left_bottom);
sparrow.components.Shell.AddComponent("ErrorList",
      sparrow.right_bottom);
sparrow.components.Shell.AddComponent("ComponentSpy",
      sparrow.center);

//-- open by default last used workspace
restore_workspace = sparrow.createcomponent("restore_workspace");
sparrow.components.ProjectManager.OpenWorkspace(
      restore_workspace.GetURI());
```

Listing 5-4 Configuration script setting the environment's layout

Listing 5-4 shows a sample configuration script that deploys components to Sparrow's shell. It uses the components table to access the available Shell components and use its AddComponent function for instantiation of five graphical components, positioning them on the available docking areas of the shell.

Moreover the script creates an instance of the `restore_workspace`, discussed in section 3.4.1, and restores the most recently used workspace in the environment's project manager. It is apparent that Delta scripts are more expressive and allow further configuration of the environment, as they can access the whole API of available components, while rules rely on adaptation logic of components to configure the environment. As the `restore_workspace` component is an add-on in the environment, it is obvious that the original author of the shell was not aware of its existence, thus could not incorporate it to the shell's adaptation logic.

The two methods of adaptation are complementary and can be used in conjunction, having user profiles that contain both specification attributes and configuration scripts.

5.6 Adaptation Components Extension APIs

Table 5-1 presents the exported API of Sparrow's Shell component. It provides functions for automatic construction of the environment through profiles, creation and manipulation of contained graphical components and addition of user commands to the application main graphical action holders, the menu and tool bars.

void	**Construct**	(String profile)
Construct the Shell contents based on a user profile.		
Handle	**GetActiveProfile**	(void)
Return a handle to the profile applied on Shell.		
Handle	**AddComponent**	(String classid, int position)
Create and position a new graphical component in the environment, returning a handle to it.		
bool	**MoveComponent**	(Handle handle, int position)
Change the position of an existing component in the Shell, returning true on success.		
bool	**RemoveComponent**	(Handle handle)
Destroy an existing component of the environment, returning true on success.		
void	**ClearComponents**	(void)
Destroy all components in the Shell.		
bool	**SetStatusText**	(String msg, int pos, int sec)
Display a message in a status bar cell for the specified duration.		
String	**GetStatusText**	(int pos)
Return the text of a status bar cell.		
void	**AddUserCommand**	(String path, UserCommandDesc desc)
Insert a user command in the Shell's main menu bar and tool bars.		
void	**RemoveUserCommand**	(String path)
Remove a user command from Shell's main menu bar and tool bars.		
UserCommandDesc	**GetUserCommand**	(String path)
Return a user command description described by the given path.		
void	**ExecuteUserCommand**	(String path)
Executes a user command of the Shell.		

Table 5-1 Shell exported API

Table 5-2 illustrates the exported API of the AdaptationManager component, including functions for creation and deletion of profiles. The SelectProfile function allows change of profiles at runtime from the list of available profiles in the environment.

bool	**CreateProfile**	(String name)
Create a new profile description file.		
bool	**DeleteProfile**	(String name)
Delete a profile from the environment.		
void	**SelectProfile**	(String name)
Select and activate a user profile.		
StringList	**GetProfiles**	(void)
Return a list of available profiles.		
void	**Refresh**	(void)
Re-scan the file directories for profiles.		

Table 5-2 Adaptation Manager exported API

6. Interactive Project Manager

6.1 Technical Approach

The organization of source their storage and build system are responsibilities of a project management module in a development environment. The incorporated project manager of Sparrow is implemented as a framework component in the environment. The concept of framework components has been discussed at Software Frameworks in section 1.1.3. A generic skeleton is defined for the general problem of project management in the environment, allowing specialization to a concrete problem, the Delta build system. Specialization is succeeded by supplying components that implement interfaces predefined by the framework. In this chapter we will discuss the project manager skeleton and the implemented components, which instantiate project manager for Delta.

The ProjectManager, seen in Figure 6-1, component provides a graphical window with a tree-view and a toolbar that presents the structure of a workspace. In essence the ProjectManager is a component container and the tree-view a hierarchical view of its contained components. The contained components are organized in a hierarchical structure defining projects, sub-projects and resource files. They should derive from the TreeItemComponent or just implement its interface in order to be displayed in the tree-view. Table 6-3 shows the interface of TreeItemComponent.

The interface allows components to define their eligible child components, defining the structure of the tree's containment hierarchy on-the-fly. There is no constraint about the depth of hierarchy, which could be from entirely flat to an arbitrary level of depth. The organization is handled entirely by the components themselves. ProjectManger component provides merely a presentation framework of the structure. The toolbar attached to the window is used for allowing contained components to present their set

of commands to the user and is context sensitive updating on selection of a component from the tree-view. Components export those actions as user commands in the environment and the ProjectManager collects them to build the toolbar.

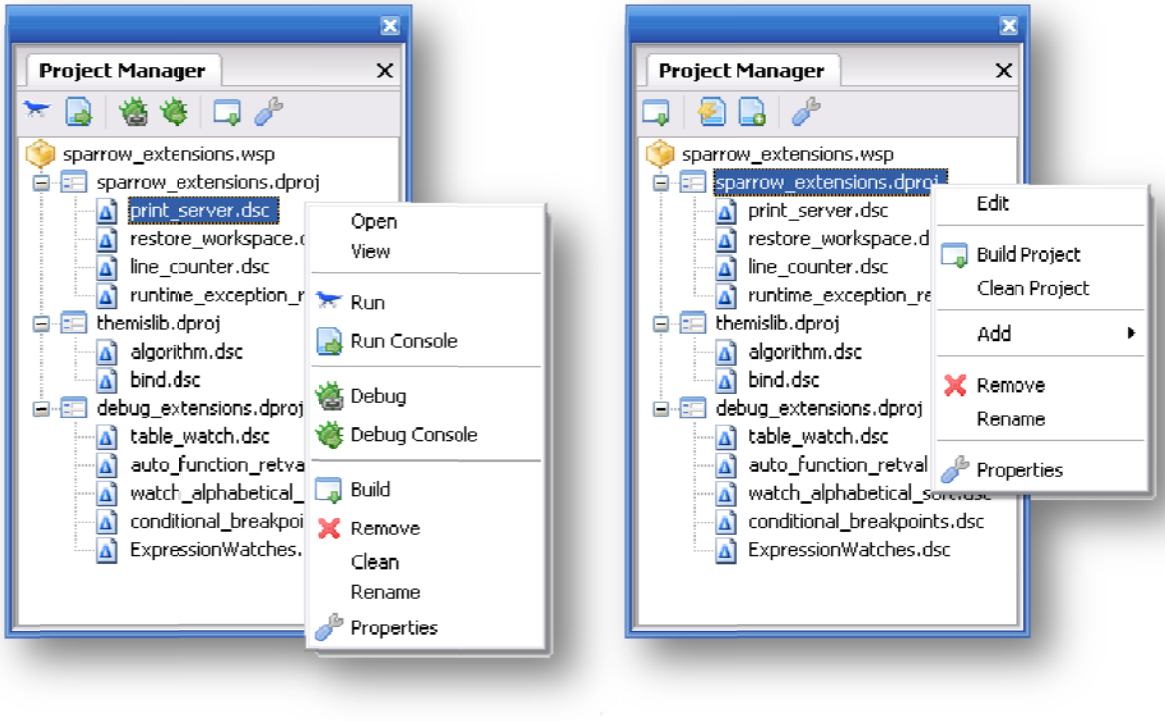

Figure 6-1 Project Manager window

6.2 Project Element Interface

For Sparrow we define four tree item components for the organization of projects. The conceptual organization is a workspace which includes several projects. Each project may contain one or more subprojects and or resource files. The resource files are Delta script files or generic files (text files, bitmaps, etc.) that the programmer may want to include to its project. The nesting of projects enables better organization of scripts and resources to the programmer.

Saving, loading and building of tree-elements are recursive operations, propagating the operation towards the leaves. For instance building a workspace translates to the workspace component propagating the command to its immediate children, the projects, which in turn propagate it to their children, subprojects or scripts.

In the following sections we will discuss the responsibilities of each the implemented component element that define Sparrow's project management facility.

6.2.1 Workspace Component

The workspace is the root of the displayed tree-view with projects as its immediate children. A workspace is actually a collection of projects that are aggregated under the some context. Workspace encapsulates the concepts of file loading, saving building and execution of resource. Workspace's children, being components as well, should comply with an interface for the operations described.

The workspace is not aware how its resources implement the operations it defines. Consequently it poses no constraints to the types of resources it accepts as its children. It may be a workspace where Delta scripts and C++ source files coexists (potentially in different projects), each building itself in its own accord without any modification to the workspace component. The workspace is only responsible for propagating the build command, not implement it.

6.2.2 Project Component

Projects are a layer below the workspace in Sparrow, aggregating resources in logical units for compilation. Like the workspace they are a layer for propagating commands to their children, being unaware of their implementation and type. The major differences between projects and workspaces are that; a workspace is unique in a project manager and workspaces are not allowed to have scripts as immediate children.

As projects and workspaces are not aware of their children until runtime there should be an infrastructure for registering eligible children for each node in the project tree. In Table 6-3 the functions that allow runtime registration of children types are presented, as part of the TreeItemComponent. Based on this information a project node is able to provide to the user a dialog of allowed children components for the project hierarchy as illustrated in Figure 6-2.

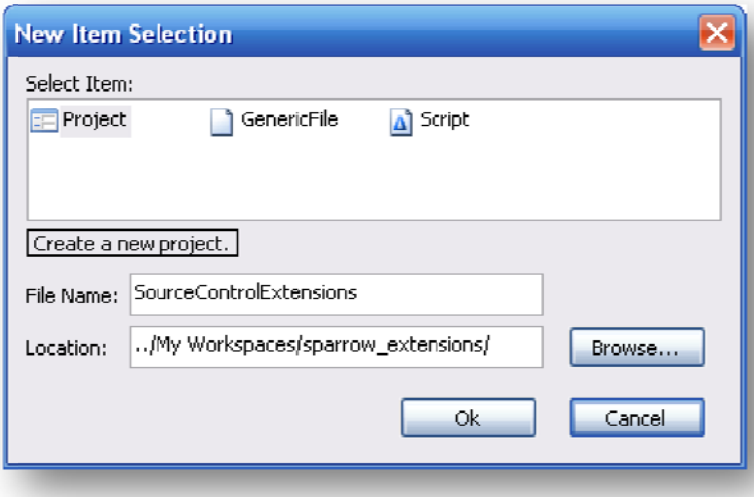

Figure 6-2 New project item dialog

Project, GenericFile and Script components have all registered as tree-node children of the project component; thus they appear at the creation resource dialog of the project.

6.2.3 Resource Components

There are, currently, two provided resource components exist in Sparrow. The first is a GenericFile component which allows any file type to be included in a project, ignoring the incoming build and execution commands from its parent components. The component is included to allow programmers load in the environment generic data files

that can be edited inside the environment. The Script component, which inherits from the GenericFile, is a representation of a Delta script source file in the project manager. It is a leaf node in the tree defining the methods for building, running and debugging a Delta script in the environment.

The Script component chooses the component that can be used to build and run the wrapped source file. As a result, in order to support another language in the workspace it would be sufficient to provide a new tree item component which would compile its wrapped resource with another compiler, having multiple languages seamlessly coexist in a workspace.

6.3 Build Process

As mentioned in previous sections the build process is recursive in the hierarchical structure of the workspace. The workspace is compiled following a depth first iteration of the tree. The Script component calls the Delta compiler for compilation of the scripts. The compiler is spawned as a console process which has its output redirected to the parent process, Sparrow IDE. The compilation results are forwarded to the Output window of the environment and the error list component. Both windows are shown in Figure 6-3, at the window's bottom pane.

The output window, on the bottom left pane, displays directly the output messages from the Delta compiler process, unfiltered to the user. On the other hand, the error list parses compiler's messages, filtering those that are errors and warnings. It distinctively marks each message according to its type, error or warning, and prints an aggregate list of messages from the workspace. The error list offers a less clattered view to the programmer. Furthermore, both tools allow navigation of the user, to compile errors and warning, by selecting the line where the message is presented. The source file opens in a source editor if not already present and is focused on the reported line of message.

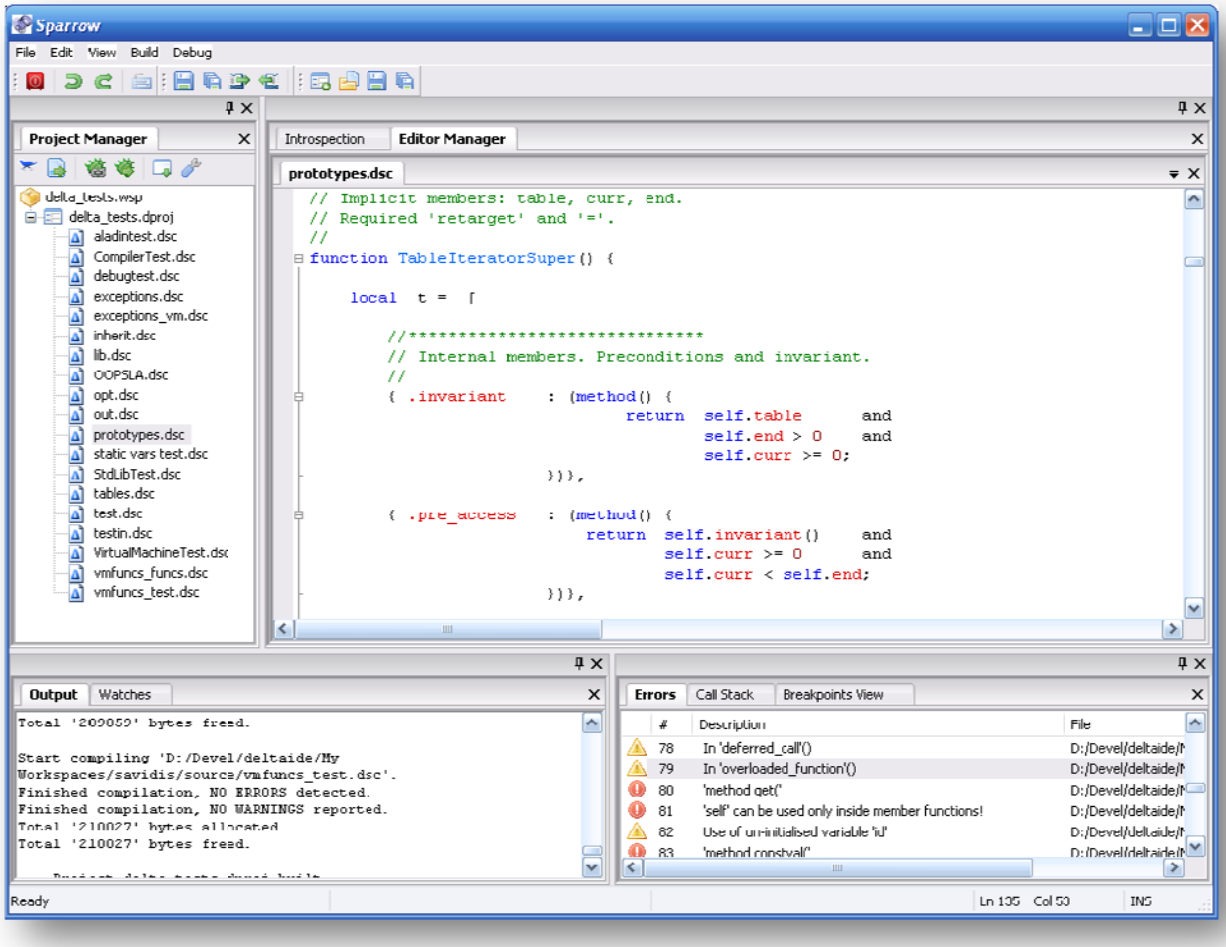

Figure 6-3 Output and Errors windows during workspace build

6.4 Project Manager Components Extension APIs

Table 6-1 presents the exported API of the DeltaCompiler component which wraps the functionality the compiler of the Delta language. The compiler concept requires a small set of functions to be implemented by the component provider and is abstract enough to comply with most compilers' requirements.

void	**Compile**	(String uri)
Compiles a resource located in the given URI.		
void	**Clean**	(String uri)
Cleans the intermediate and target files by for the compilation of resource specified by the given URI.		
void	**SetOutputPath**	(String path)
Specifies the output path of the intermediate and target objects of the compilation process.		
String	**GetOutputPath**	(void)
Returns the output path of the intermediate and target objects of the compilation process.		
void	**sigCompileDone**	(String uri)
Signal fired by the completion of a script's compilation.		
String	**sigCleanDone**	(String uri)
Signal fired by the completion of a script's clean process.		

Table 6-1 DeltaCompiler exported API and signals

Table 6-2 presents the exported API of the ProjectManager framework component. In its API the project manager has only the notion of the workspace. The rest components beyond the workspace are handled as black boxes to it.

bool	**OpenWorkspace**	(String uri)
Load a workspace from file.		
Handle	**GetWorkspace**	(void)
Return a handle to the currently active workspace.		
void	**NewWorkspace**	(void)
Create a new workspace in the environment.		

void **OpenWorkspaceDialog** (void)
Choose an existing workspace to open from a file-open dialog.

void **SaveWorkspace** (void)
Save the currently active workspace on disk.

void **SaveAll** (void)
Save recursively all resource in the workspace.

void **CloseWorkspace** (void)
Close the currently open workspace.

bool **AddComponent** (Handle parent, Handle handle)
Insert a component in the project tree view under the specified parent.

bool **RemoveComponent** (Handle handle)
Remove a component from the tree view.

bool **RenameComponent** (Handle handle, String name)
Rename an already existing component in the tree view.

HandleList **GetResources** (String type)
Return a list of handles of all components of a given type in the tree view.

Table 6-2 ProjectManager export API

In Table 6-3 the minimum required API by a component, which can be placed inside the component tree hierarchy, is presented. RegisterChildType function allows tree component to define constraints about their placement in the hierarchy. This registration method allows, for instance a project, to learn at runtime what types of resources are available without needing to be hard-coded types in its code.

void **RegisterChildType** (String parented, String classId)
Register a component class as an eligible child of the parent component class in the tree structure.

int **UnregisterChildType** (void)
Unregister a component class from being an eligible child of the parent component class in the tree structure.

StringList **GetChildrenTypes** (String parentId)
Return the list of component ids that are eligible children of the parent component.

String	**GetName**	(void)
Return the name of the resource.		
String	**GetURI**	(void)
Return the URI of the resource.		
void	**SetName**	(String name)
Set the name of the resource.		
void	**SetURI**	(String uri)
Set the URI of the resource.		
Handle	**GetChild**	(String uri)
Return a handle to a child component given its URI.		
HandleList	**CollectChildren**	(String type)
Return a list of handles of all children components of the specified type.		

Table 6-3 TreeItemComponent exported API

Finally Table 6-4 presents the API required by the workspace and consequently the project components to be implement by their child components.

bool	**Load**	(String uri)
Load resource from file.		
bool	**Save**	(void)
Save resource to disk.		
bool	**SaveAll**	(void)
Save resource and its children recursively.		
void	**Build**	(void)
Build resource.		
void	**Clean**	(void)
Clean output files created during building of the resource.		
void	**Run**	(void)
Execute target.		
void	**Debug**	(void)
Execute target in debug mode.		

Table 6-4 Required exported API from project resource components

7. Conclusions and Future Work

7.1 Future Work

Sparrow has set the main platform for a Delta development environment. Scripts in Delta can manipulate and extend the IDE to a varied degree providing many facilities, not originally implemented. Our future plans include further extensions on top of provided debugging utilities for customized views of language constructs. It is interesting to try experiment and customize the presentation of elements during the debugging session to deliver the most suitable approach for each developer.

The most appealing potential we wish to pursue is the deployment of the Sparrow in an existing external environment. We intend to incorporate Sparrow in UnderGo, a game editor supporting a game engine developed internally in HCI lab, ICS-FORTH. Currently UnderGo implements part of a Delta IDE's functionality internally for editing and debugging of Delta scripts that are used for programming game logic over the engine. The embedded editor of the application can be replaced by a subsystem that will communicate via the deployment API with Sparrow, which will manipulate game scripts. The structure of script projects per game level can be organized by UnderGo, using Sparrow's project manager to form logical directory hierarchies.

The adaptation infrastructure of the environment can provide a platform suitable for game development, in collaboration with the game editor. Furthermore extensions can be created that would aid the development of game scripts in Sparrow, potentially operating on more file types than Delta scripts, like content description files.

7.2 Conclusions

In this thesis we have presented a part of Sparrow IDE, a meta IDE for the Delta language. Our target was not making an IDE as such, but an open tool that as a starting point offers IDE functionality. It offers circularity which allows self extension, and is meta, which allows third-party customization and deployment. On the other hand, Sparrow IDE had to be a working Delta IDE so we have developed appropriate features as components; project manager, source level debugger.

In addition we have demonstrated the adaptation features of the environment that allow it to be custom-tailored to the user's requirements. We have also discussed how circular extensibility, adaptation and deployment of Sparrow to external tools give Sparrow the meta characteristic.

Through the process we pursued to design and construct an open platform to extensions. We avoided drawing hard links between system's modules, by componentizing the environment. The investment on building an open component-based platform was a real payoff. In the course of our development we appreciated that our modular design could support substitution of components at runtime. Dynamic component substitution allowed us to create an environment that can adapt at runtime to user's requirements.

As a final note we should mention an emerging pattern during the end of the development. Once the basic environment was built and the Sparrow IDE core become stable, we tended to increasingly replace native C++ code with Delta code, reimplementing components as Delta extensions. The circularity helped us incorporate and maintain features faster in Delta than in C++.

8. Bibliography

1. *Dynamic Imperative Languages for Runtime Extensible Semantics and Polymorphic Meta-Programming.* **Savidis, A.** 2005. RISE. pp. 113-128.

2. **Martin, A. and Cardelli, L.** *A Theory of Objects.* s.l. : Springer-Verlag.

3. *An Efficient Implementation of SELF - a Dynamically-Typed Object-Oriented Language Based on Prototypes.* **Chambers, C., Ungar, D. and Lee, E.** 1989. OOPSLA. pp. 49-70.

4. **Ousterhout, J. K.** Scripting: Higher-Level Programming for the 21st Century. *IEEE Computer.* 3 31, 1998, pp. 23-30.

5. **Abrahams, D. and Gurtovoy, A.** *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond.* Reading, MA : Addison-Wesley, 2004.

6. *Unidraw: a framework for building domain-specific graphical editors.* **Vlissides, J. M. and Linton, M. A.** 3, s.l. : ACM Transactions of Information Systems, Vol. 8, pp. 237-268.

7. *A Decision-making Specification Language for Verifiable User-interface Adaptation Logic.* **Savidis, A., Antona, M. and Stephanidis, C.** 2005, International Journal of Software Engineering and Knowledge Engineering, pp. 1063-1094.

8. **Gamma, E., et al.** *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA : Addison-Wesley, 1995.

9. **Corporation, Microsoft.** *OLE 2 Programmer's Reference: Creating Programmable Applications with OLE Automation v. 2, Programmer's Reference Library.* s.l. : Microsoft Press, 1993.

10. *Unified user interface development: the software engineering of universally accessible interactions.* **Savidis, A. and Stephanidis, C.** s.l. : Universal Access in the Information Society 3, 2004, HCI, pp. 165-193.

11. *Dynamic Software Assembly for Automatic Deployment-oriented Adaptation.* **Savidis, A.** 2005, Electr. Notes Theor. Comput. Sci. , pp. 207-211.

12. **Heineman, G. T. and Councill, W. T.** *Component-Based Software Engineering: Putting the Pieces Together.* Reading, MA : Addison-Wesley Professional, 2001.

13. **Meyer, B.** *Object-Oriented Software Construction. 2nd ed.* s.l. : Prentice Hall, 1997.

14. OMG-CORBA. [Online] http://www.omg.org/.

15. **Box, D.** *Essential COM.* Reading, MA : Addison-Wesley, 1998.

16. Microsoft Visual Studio. [Online] http://msdn2.microsoft.com/en-us/vstudio/default.aspx.

17. *Eclipse - An Open Source Platform for the Next Generation of Development Tools.* **Weinand, A.** 2003. NetObjectDays.

18. *Open, extensible dynamic programming systems.* **Piumarta, Ian.** s.l. : Dynamic Languages Symposium, 2006.