University of Crete
Computer Science Department

CIRCULAR META-IDE FOR THE DELTA LANGUAGE:
DYNAMIC EXTENSIBILITY, REMOTE DEPLOYMENT, INTERACTIVE INTROSPECTION AND SYNTAX DIRECTED EDITOR

by

YANNIS GEORGALIS

MASTER'S THESIS

Heraklion, October 2007

ii

University of Crete
Computer Science Department

CIRCULAR META-IDE FOR THE DELTA LANGUAGE:
DYNAMIC EXTENSIBILITY, REMOTE DEPLOYMENT, INTERACTIVE INTROSPECTION AND SYNTAX DIRECTED EDITOR

by

YANNIS GEORGALIS

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Author: _____
Yannis Georgalis

Board of enquiry:

Supervisor _____
Constantine Stephanidis, Professor

Supervisor _____
Anthony Savidis, Associate Professor

Member _____
Evangelos Markatos, Professor

Member _____
Mema Roussopoulos, Assistant Professor

*The board of enquiry judged that the present Thesis is also given the characterization
With Distinction.

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

Heraklion, October 2007

Abstract

Following programming languages, Integrated Development Environments (IDEs) are considered as the next decisive factor for effective software development, playing a critical role in the software lifecycle, especially when it targets medium-to-large-scale systems. In this context, the subject of this Thesis is Sparrow; an IDE for the dynamic, object-based programming language – Delta. Sparrow was developed with the following two key objectives: (a) to support extensibility of features, allowing such extensions to be developed using Sparrow, i.e. it is a circular IDE, and (b) to facilitate open deployment by third parties to build domain-oriented IDEs, i.e. it is a meta-IDE.

In this Thesis, the design and implementation of a large part of Sparrow has been carried out – corresponding roughly to half of the system's implementation, – addressing the following issues: (a) the implementation of the basic component framework for extensibility, enabling developers dynamically introduce IDE components, (b) the implementation of the mechanism for remote deployment, enabling third-party applications dynamically utilize the IDE in a domain specific manner, (c) the implementation of a component introspection User Interface, enabling users interactively review and invoke the underlying functionality of all IDE components, and (d) the implementation of the source code editor supporting real-time, true syntax highlighting during editing, relying on quick incremental parsing particularly suited to the Delta language.

The work reported in this Thesis enabled the Sparrow IDE to play the role of an open platform capable of dynamically hosting IDE functionality, reflecting the *tabula rasa* concept. Along these lines, we expect future IDEs to move towards these directions, delivering more flexible and open infrastructures by enabling users introduce extensions and customizations reflecting their individual programming habits or any emerging programming techniques.

Περίληψη

Μετά τις γλώσσες προγραμματισμού, τα Ολοκληρωμένα Περιβάλλοντα Ανάπτυξης (Integrated Development Environments - IDEs) θεωρούνται ο σημαντικότερος παράγοντας για την ανάπτυξη λογισμικού, παίζοντας κρίσιμο ρόλο στον κύκλο ανάπτυξης προγραμμάτων, ιδιαίτερα δε για τα μεσαίας και μεγάλης κλίμακας συστήματα. Σε αυτό το πλαίσιο, το θέμα της παρούσας εργασίας είναι το σύστημα Sparrow, ένα IDE για τη δυναμική οντοκεντρική γλώσσα Delta. Το Sparrow κατασκευάστηκε ακολουθώντας δύο κύριους στόχους: (α) να υποστηρίζει επεκτασιμότητα των λειτουργιών του, επιτρέποντας να αναπτυχθούν οι επεκτάσεις αυτές χρησιμοποιώντας το ίδιο το Sparrow, δηλαδή είναι ένα κυκλικό IDE, και (β) να υποστηρίζει τη χρήση του από τρίτα συστήματα διευκολύνοντας τη δημιουργία IDEs εξειδικευμένων στο εκάστοτε πεδίο εφαρμογών, δηλαδή είναι ένα μετα-IDE.

Στο πλαίσιο αυτής της εργασίας, αναπτύχθηκε ένα μεγάλο μέρος του Sparrow, που αντιπροσωπεύει περίπου το ήμισι της υλοποίησης του όλου συστήματος, και αφορά στα παρακάτω ζητήματα: (α) την υλοποίηση της βασικής δομής διαχείρισης τμημάτων λογισμικού που υποστηρίζει την επεκτασιμότητα του συστήματος, η οποία καθιστά δυνατή τη δυναμική εισαγωγή και χρήση των τμημάτων από τους προγραμματιστές, (β) την υλοποίηση του μηχανισμού ελέγχου μέσω δικτύου, ο οποίος επιτρέπει σε εξωτερικές εφαρμογές να χρησιμοποιούν δυναμικά το IDE ως τμήμα, με τρόπο που εξαρτάται από το εκάστοτε πεδίο εφαρμογών, (γ) την υλοποίηση διεπαφής ενδοσκόπησης των τμημάτων κώδικα, η οποία επιτρέπει στους προγραμματιστές να βλέπουν και να καλούν τις λειτουργίες των τμημάτων του IDE κατά τη διάρκεια της χρήσης του, και (δ) την υλοποίηση του συντάκτη κώδικα, ο οποίος μπορεί να παρουσιάζει σε πραγματικό χρόνο με γραφικό τρόπο τα τμήματα του πηγαίου κώδικα σύμφωνα με το συντακτικό της γλώσσας, υλοποιώντας μία μέθοδο γρήγορης, αυξητικής συντακτικής ανάλυσης ειδικά σχεδιασμένης για τη γλώσσα Delta.

Η παρούσα εργασία κατέστησε εφικτή την ανάπτυξη του Sparrow ως μία ανοικτή και επεκτάσιμη πλατφόρμα λογισμικού, ακολουθώντας τη φιλοσοφία *tabula rasa*, ώστε

να υποστηρίζει ευέλικτα τη λειτουργικότητα ενός IDE υλοποιώντας ένα γενικό αρχιτεκτονικό πλαίσιο που υποστηρίζει τη δυναμική συρραφή των λειτουργικών τμημάτων. Σε αυτές τις γραμμές, αναμένουμε τα μελλοντικά IDEs να κινηθούν σε παρόμοιες κατευθύνσεις, προσφέροντας ακόμη πιο ευέλικτες και ανοικτές υποδομές, επιτρέποντας στους προγραμματιστές να εισάγουν επεκτάσεις και προσαρμογές σύμφωνα με τις ιδιαίτερές τους προγραμματιστικές συνήθειες καθώς και τις εκάστοτε αναδυόμενες προγραμματιστικές τεχνικές.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τους επόπτες της μεταπτυχιακής μου εργασίας Αντώνιο Σαββίδη και Κωνσταντίνο Στεφανίδη για την συνεχή καθοδήγηση και υποστήριξή τους τα τελευταία τεσσεράμισι χρόνια στο πλαίσιο της συνεργασίας μας στο Εργαστήριο Επικοινωνίας Ανθρώπου-Υπολογιστή, του Ινστιτούτου Πληροφορικής του Ιδρύματος Τεχνολογίας και Έρευνας.

Επίσης, θα ήθελα να ευχαριστήσω τον Θεμιστοκλή Μπουρδένα με τον οποίο είχα τη χαρά να συνεργάζομαι συνολικά τρία χρόνια, ιδιαίτερα δε τους τελευταίους επτά μήνες στο πλαίσιο της εκπόνησης των μεταπτυχιακών μας εργασιών που είχαν ως κοινό στόχο την ανάπτυξη του Sparrow IDE.

Ευχαριστώ, τέλος, την οικογένειά μου και τους φίλους μου που με υπέφεραν και με στήριξαν όλα αυτά τα χρόνια.

Table of contents

List of figures

List of tables

1. Introduction

Computer programs follow completely different architectural and implementation strategies in relation to the problem being solved, the available resources, and the way users interact with them. This multi-modality that inevitably characterizes software systems not only increases their complexity, but also impedes the formalization of concrete guidelines and "recipes" for approaching the construction of a program.

The plethora of available programming languages and development tools clearly reflect the aforementioned lack of formalization of the development process. Whereas the advancements in high-level languages have undisputedly allowed software systems to become much more sophisticated, Integrated Development Environments have enabled programmers to produce more robust programs in a smaller time frame. An Integrated Development Environment (IDE) is basically a program that assists the process of software authoring by (a) disengaging the programmer from source code maintenance operations, (b) visually annotating and validating the syntactical structures of a program, (c) aiding the debugging process, and (d) automating some aspects of the code manipulation operations.

The final product of this project – dubbed Sparrow – is an IDE for the dynamic object-based programming language, Delta. The project's goal was not only to create a full fledged development environment for the Delta language, but also to explore the usage of various programming techniques for implementing a dynamically configurable, remotely deployable, and extensible software platform.

1.1 Objectives

An IDE, besides being a program that aids the developer by automating tedious programming tasks, ought to provide a concrete platform on which developers can build custom tool-chains and extend its functionality. This was viewed as the most critical factor when designing Sparrow. The provision of a sensible, intuitive meta-

development platform that can leverage the effectiveness of the offered facilities is missing from most contemporary IDEs.

1.1.1 Common features

Sparrow aims to be a full-fledged IDE. Hence, its features that are similar to the ones offered by many existing Integrated Development Environments are the following:

- Source code editor with highlighting support that annotates and validates Delta's syntactic constructs on-the-fly
- Workspace manager for managing the collection of source files that comprise a Delta program and their properties
- Source-level debugger for the Delta language
- Extensibility interface for extending all aspects of the IDE's functionality through both C++ and Delta languages
- Deployment interface for accessing part of the IDE's functionality from other programs
- Support for a multi-lingual interface

1.1.2 Novel features

This Thesis focuses on four aspects of Sparrow, namely: extensibility through its component-based architecture, remote deployment capabilities, interactive introspection facility, and Delta source code editor. These subsystems correspond roughly to half of the IDE's code volume and functionality.

Sparrow offers a concrete component-based architecture that clearly separates the different parts of the IDE's user interface and functionality, while automatically exposing their facilities through its extensibility Application Programming Interface (API). This explicit componentization of Sparrow not only enhances its maintainability, but also allows for well-defined and straightforward extensions to the IDE by either activating different components at run-time or enhancing those already available through Delta or native extensions. Additionally, Sparrow offers a

centralized Undo subsystem that simplifies to a large degree the provision of Undo/Redo functionality by components.

To make it easier for third-party applications to deploy the IDE, Sparrow provides a mechanism that allows an arbitrary number of processes – that can even run on remote machines – to utilize the functionality that is exposed by the available components.

Additionally, Sparrow offers a graphical component that is able to extract and display all the introspection data that are encapsulated into its components. This tool offers to the extension programmer a comprehensive reference of (a) all the supported functions, (b) the components' metadata, and (c) their active instances. Through the provided interface, programmers are able to interactively manipulate many aspects of the IDE on-the-fly.

Lastly, Sparrow's source code editor features a complete Delta language parser that retains the whole syntactical structure of Delta programs and exposes it to the extension scripts. The supplied Delta parser is able to parse the target programs incrementally. A change in the target program will trigger the reevaluation of only the parts of the program that are affected by this change rather than the whole source file. The facilities of the editor that are based on the internal representation of the edited program include: (a) syntax highlighting and code folding, (b) visualization of the program, (c) syntax validation, (d) automatic symbol completion, and (e) informative tooltips on language constructs.

1.2 Architecture

At the most basic level, Sparrow is comprised of a set of loosely-coupled components that communicate with each other through message passing. In this sense, Sparrow follows a *tabula rasa* approach; the core of the IDE, its component system, provides the basic functionality in order to accommodate the various components that infuse the functionality and the graphical user interface to the IDE. Sparrow's components

can be implemented in either C++ (Sparrow's native language) or Delta. A schematic representation of this notion can be seen in Figure 1.
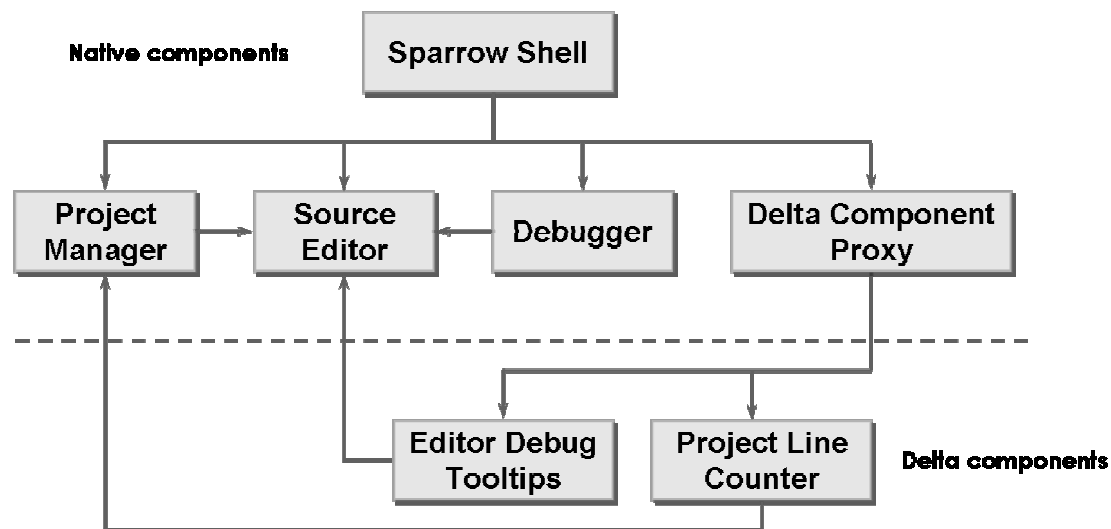
Figure 1 - Sparrow as a collection of components

Essentially, Sparrow is bootstrapped by the "Shell" component which constitutes the basic skeleton that initially instantiates the desired components – read from a configuration file. This skeleton also implements a graphical frame under which the top-level components, such as the editor and the project manager, present their interface. Delta components, which are indistinguishable from native components, are managed by a separate component, dubbed "Delta Component Proxy."

1.2.1 Circularity

Sparrow's circularity refers to its ability to incorporate in its environment the Delta components that are developed in the IDE itself. Generally, the facilities that enable the implementation and usage of C++ components are referred to as the *native* extensibility layer, and, correspondingly, the facilities that allow the deployment of Delta components as the *circular* extensibility layer (see Figure 2.)

Circularity, in this context, is different from the circularity offered by environments that target the same language they are built in. It is evident that any program can be extended through the language it is written in. Sparrow, however, while targeting the

4

Delta language, is programmed in C++. Under this realization, it can be inferred that the Sparrow platform offers *true* circularity.
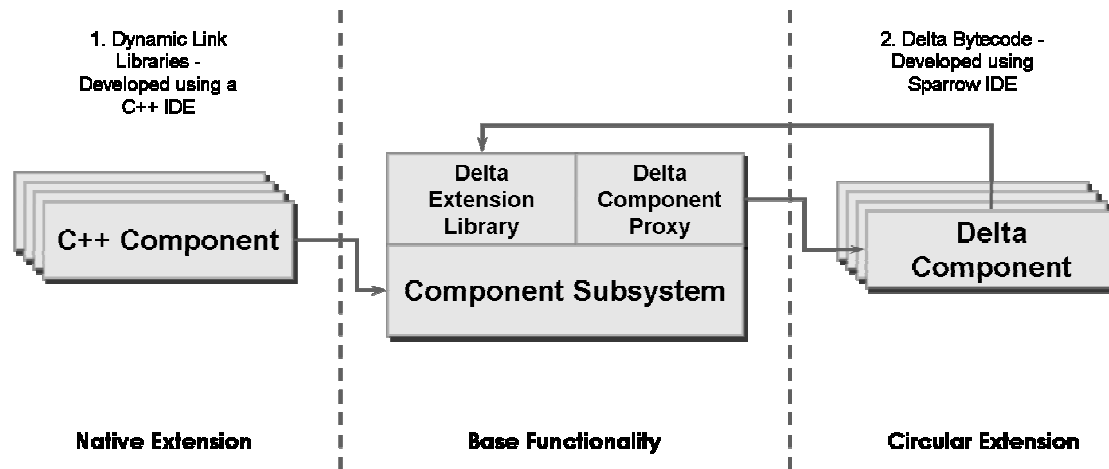
Figure 2 - Native and Circular extension layers in Sparrow

1.2.2 Meta

The *meta* notion of Sparrow emphasizes its facilitation for open deployment by third-party tools that can customize and extend the platform – essentially producing a development environment that is better suited to their problem domain. The idea is that for applications relying on the Delta language (e.g. games, mobile applications, etc.,) the IDE should deliver the basic programming facilities, while enabling the incorporation of functionality through the development of extension components or customization of the existing ones. This constitutes the driving factor behind Sparrow, whose architecture evolved, or rather *intelligently designed*, around this notion.

For this purpose, Sparrow (a) enables its remote deployment by third-party applications, (b) allows every integrated component to be replaced as long as it obeys the original API (runtime consistency), and provides related semantic behavior (semantic consistency), and (c) allows the incorporation of new components that can extend the functionality of the existing ones. Additionally, some integrated components (e.g. the source editor,) support their own configuration switches and APIs so that they can be extended orthogonally to the Sparrow platform.

Categorizing the aforementioned extension mechanisms, it can be inferred that Sparrow supports two types of domain-specific extensions:

- Horizontal extensions
- Vertical extensions

This principle is outlined in Figure 3. Generally, vertical extensions refer to the incorporation of new components that basically introduce new functionality to the IDE; whereas substitutions or extensions of existing components are regarded as horizontal extensions.
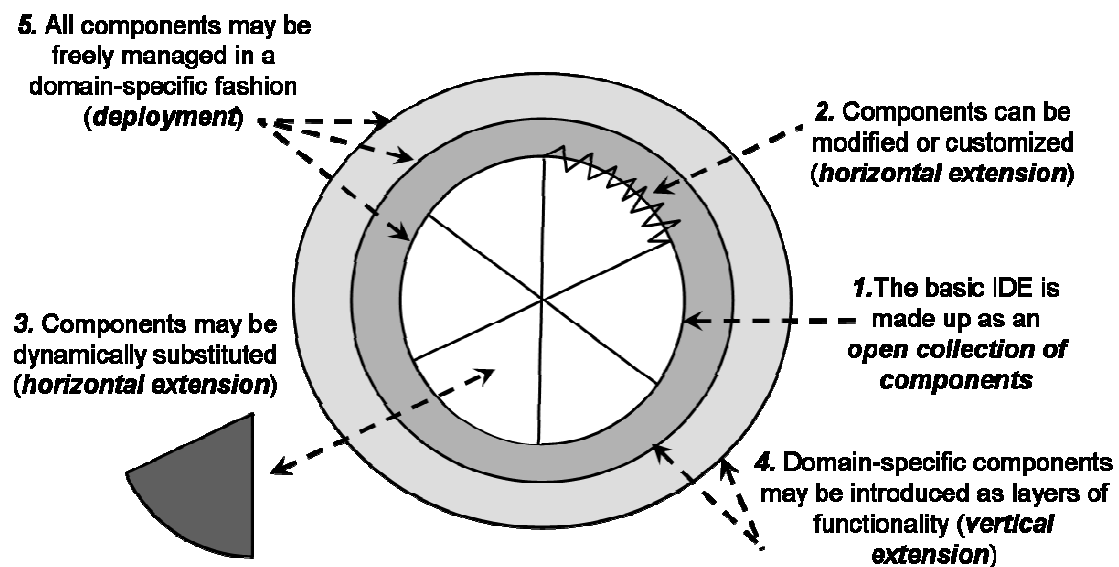
Figure 3 - Horizontal and Vertical extensibility in Sparrow

1.3 Methodology

Unarguably, the implementation of an IDE constitutes a very large development effort. Under this realization, the choice of the programming libraries and techniques was very important for the successful completion of the project within the bounds of the desired time-frame. Apart from Delta, through which Sparrow can be extended, the core development language was decided to be C++. C++ was chosen for the project because of (a) its ubiquity, (b) the large number and high quality of its third party libraries, (c) its support for a variety of programming paradigms (especially Generic programming [2] and Object Oriented programming [18],) and (d) because of

the fact that Delta itself is written in C++, and therefore can be easily deployed in C++-based applications.

For the development of Sparrow, the Boost libraries [4] were used, along with the Standard Template Library (STL,) while the wxWidgets library [29] was used for the implementation of the Graphical User Interface (GUI.)
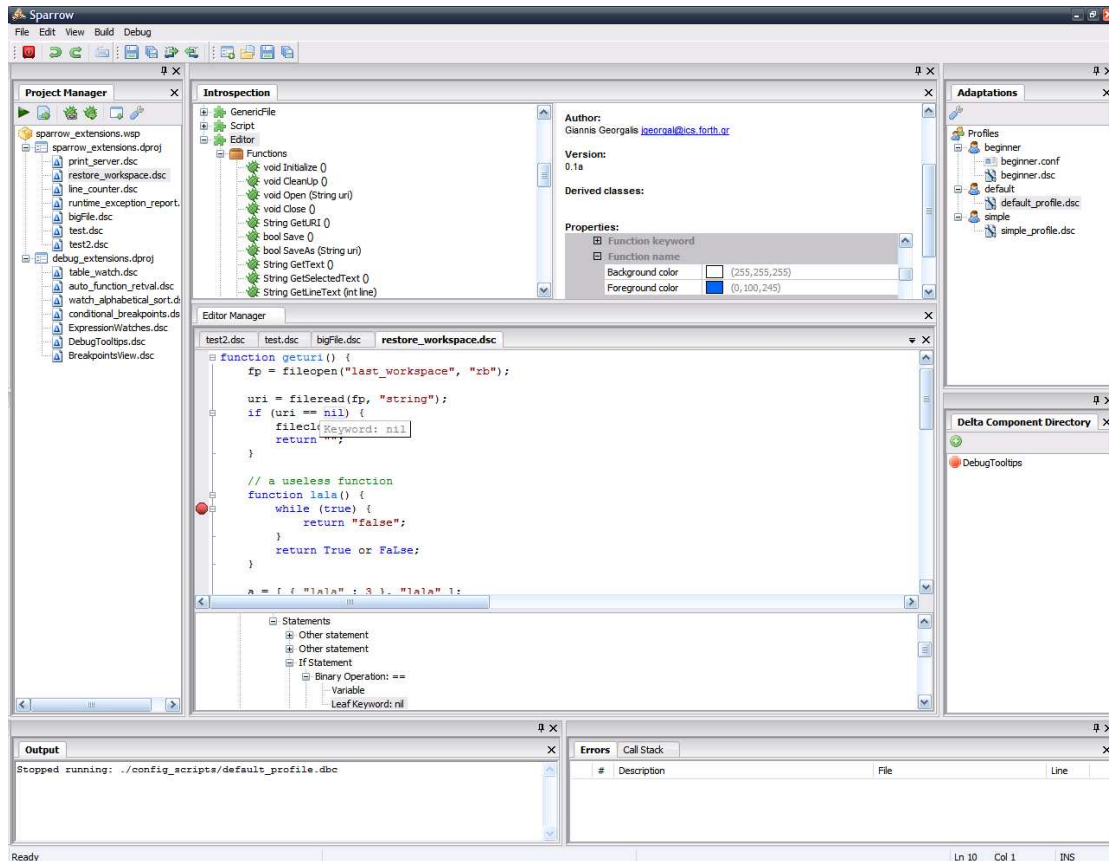
Figure 4 - Screenshot of Sparrow with various key components active

2. Related work

2.1 The Delta Language

The Delta programming language [23] is an imperative scripting language that encompasses (a) dynamically typed variables, (b) runtime classes, (c) functions as first-class values, (d) unnamed functions, (e) dynamic handling of actual arguments,

and (f) extensible operator semantics. The aforementioned features are available in most modern dynamic languages. However, Delta extends these features by introducing:

- Prototypes with member functions being independent callable first-class values, as atomic pairs holding both the function address and the alterable owner instance
- Dynamic inheritance, having exclusively runtime semantics, in comparison to the traditional compile-time inheritance operators
- An enhanced operator overloading technique

In the Delta language, prototypes are runtime class values, from which instances are dynamically produced through replication. In this context, following the recipe of existing dynamic languages, object classes never appear within the source code in the form of compile-time manifested types, but only as first-class runtime values called prototypes. The main characteristic of prototypes in the Delta language is that they are essentially associative table objects. Having no prototype-specialized compile-time or run-time semantics, prototypes are normal object instances chosen by programmers to play the role of class-instance generators, thus, they are effectively a design pattern [8] combined with a deployment contract.

In Delta, inheritance is a runtime function applied to instances, establishing an augmented member-binding context for derived instances. The metaphoric *isA* connotation of base and derived classes is not entirely adopted in Delta, since *inherit(x, y)* does not state that *x isA y*, neither that *x* depends implementation-wise on *y*; it only defines augmented member binding for both *x* and *y*, i.e. if a member requested for *x* or *y* is not found in *x* (derived,) then it is searched in *y* (base.)

Additionally, in Delta, the semantics of all binary operators are dynamically extensible for table object instances through the following implementation technique:

- For binary operators, if a member of a table instance *t1* is named *op* and is actually a function *f*, the result of the evaluation *t1 op t2* is *f(t1, t2)*. Otherwise, the original semantics of *t1 op t2* are applied

- For unary operators, if a member of a table instance *t1* is named *op* and is actually a function *f*, the result of the evaluation *op t1* is *f(t1)*. Otherwise, the original semantics of *op t1* are applied

2.2 Existing IDEs

There is a very large number of high quality Integrated Development Environments available. Nonetheless, they all follow similar patterns and interaction metaphors. Hence, their supported functionality has many common features, and it generally includes the following:

- Source code highlighting and completion
- Automation of source code maintenance
- Source-level debugger
- Extensibility interface for adding or substituting functionality and automating common tasks

In addition, some of the contemporary IDEs support:

- Remote (inter-process) deployment interface
- Refactoring tools for the target languages
- Highly configurable user interface

The overview of the IDEs presented in this section is based on the set of features that are related to the ones considered in this Thesis for Sparrow. Specifically, the following characteristics are examined:

- Extensibility, which refers to how easily the IDE can be extended to incorporate additional functionality
- Deployability, which refers to the level of the IDE's functionality which is exposed to third party applications
- Syntax analysis, which refers to the level of assistance the IDE provides to the programmer in relation to the syntactical structure of the supported languages

The majority of contemporary IDEs support more than one language under their interface. Thus, the aforementioned characteristics are considered for the main language of each IDE, that is, the most supported language.

2.2.1 Visual Studio

Microsoft's Visual Studio [13] for the Windows operating system constitutes the primary development tool for the company's .NET platform. As such, many different languages are supported and more are being added – or at least announced – as incremental updates to the platform. In addition to the .NET environment, Visual Studio also targets the native Windows platform through the Visual C++ tool-chain; nonetheless, the .NET languages are better supported. Therefore, the most popular .NET language, C#, is considered as the IDE's main language.

Visual Studio is built on top of the COM [6] component framework. Extensions to it come in the form of macros, add-ins, and packages. Macros represent repeatable tasks and actions that developers can record programmatically to automate common tasks. Add-ins enable languages that support COM (i.e. C++, Visual Basic and .NET languages) to be used for extending the functionality of the IDE and controlling existing Visual Studio elements. Finally, packages fully expose the platform's C++ interfaces to programmers who can use them to build complete replacements for all the elements that are available to Visual Studio. Actually, all the languages that are supported in Visual Studio are developed as packages.

Despite the fact that COM supports the remote invocation of its objects through the IDispatch interface – a technology dubbed Object Linking and Embedding (OLE) automation [11] – Visual Studio does not provide any documentation for the usage of these interfaces, making hard its deployment from other processes.

Visual Studio's editor validates and maintains the syntactic structure of the edited program by exploiting the information that is provided by the language's compiler. That enables the environment to indicate potential errors in the structure of the

program, support automatic completion for object members, and provide a set of refactoring tools.

2.2.2 Eclipse

Eclipse Foundation's Eclipse IDE [26], originally designed and implemented by IBM, aims to offer a comprehensive service platform for integrating development and deployment tools for a variety of programming languages. The Eclipse platform, however, mainly constitutes a complete IDE for the language it is written in – Java.

Eclipse employs a component framework based on the OSGi [21] specification in order to provide all of its functionality on top of its platform. Though that mechanism, Eclipse can be fully extended in the Java language as it essentially allows programmers to access the platform's components and replace them by implementing their Java abstract interfaces.

Through the mechanisms specified by the underlying OSGi standard, Eclipse can be deployed from other languages that implement the specification even when they are invoked from other processes.

Lastly, Eclipse's editor for the Java programming language utilizes the compiler to validate the edited program's syntax. By using the compiler's internal representation of the program, the editor provides refactoring tools and automatic symbol completion for Java objects.

2.2.3 KDevelop

KDE project's KDevelop [16] is the official IDE of the KDE desktop environment. As such, it is heavily based on KDE and Qt [26] technologies. KDevelop targets mainly the C++ programming language but can accommodate other languages as well.

KDevelop uses the KParts [17] framework in order to support its component-based architecture. Through this framework, the programmer can access, extend, or replace completely the existing components of the IDE in C++. In addition, Kdevelop embeds the Python [22] interpreter in order to enable the construction of extensions in the Python language.

Additionally, KDevelop uses KDE's DCop [15] technology to allow the inter-process deployment of the IDE. However, DCop does not automatically expose a component's interface; so the programmer needs to maintain a DCop interface in addition to the KPart-enabled one.

As far as the editor is concerned, KDevelop does not retain the program's structure and, thus, cannot validate the structure of the edited text. Nonetheless, it employs a lightweight C++ parser in order to extract the relevant symbols from the hosted project's source files and present them in a completion list to the programmer when needed. This approach is analogous to the approach of Visual Studio's Intellisense tool – for the Visual C++ language – whose only function is the extraction of the relevant symbols. In any case the edited program's structure is discarded as soon as the symbols are extracted.

2.2.4 Comparison

Table 1 summarizes the aforementioned traits of the featured IDEs, including Sparrow. A scale from zero to three is used to evaluate the support level of each characteristic for each of the IDEs. The attributed grades have the following meaning:

0. The feature is not implemented
1. The feature is available, but it requires substantial effort in order to be utilized
2. The feature is supported
3. The feature is supported and can be efficiently utilized

Table 1 - Comparison of IDEs

	Visual Studio	*Eclipse*	*Sparrow*	*KDevelop*
Main language	C#	Java	Delta	C++
Component framework	COM	OSGi	Sparrow component framework	KParts
Extensibility	3	3	3	3
Deployability	1	3	3	2
Syntax analysis	3	3	3	0

Comparing the evaluated IDEs with Sparrow, there are a few things worth noting. First of all, Sparrow's architecture has very similar goals and capabilities with the Eclipse IDE. They both enforce the Aristotelian *tabula rasa* concept and support extensibility and deployment efficiently and effectively. Their main difference, apart from the implementation language, is the usage of components. Whereas Eclipse uses static interfaces for enforcing a communication protocol between components, Sparrow is inherently more dynamic allowing the construction and extension of component interfaces at runtime. Additionally, while Eclipse enables other programming languages to be used for the extension of the platform only when they implement the whole OSGi specification, Sparrow requires only an inter-component proxy and a target-language library to achieve the same goal.

Secondly, there are fundamental architectural differences, as far as deployment is concerned, in Sparrow's approach compared to the approaches of Visual Studio and, especially, KDevelop. Whereas Sparrow automatically exports the interface of all its components to both other components and remote processes, Visual Studio and KDevelop utilize separate mechanisms for interface exporting in these two instances. Additionally, the differences in component usage that were outlined for Eclipse above are true for both Visual Studio and KDevelop. In fact all three systems follow similar mechanics for utilizing their components.

Lastly, Visual Studio and Eclipse, which support syntax analysis of the edited program, achieve this goal by utilizing the compiler and evaluating the whole program each time they need to construct a structured representation. In contrast,

Sparrow evaluates only the parts of the text that affect the representation and does so every time the edited program is modified in the editor.

3. Dynamic Extensibility

Sparrow's core extensibility capabilities are facilitated by its component-based architecture. By disseminating the IDE's functionality in distinct well-defined modules that expose a sensible control Application Programming Interface (API), not only enhances the maintainability and robustness of the IDE, but also provides the means to alter its functionality at runtime. Hence, dynamic extensibility in this context refers to the ability of the IDE to extend and alter its functionality at runtime by means of vertical and horizontal extensions.

In the following sub-sections, the component infrastructure of Sparrow, the facilities it provides for extensibility and the main subsystems that were built on top of it will be presented.

3.1 Components

Software components (or Components) [7] are self-contained, reusable software units that encapsulate and expose a well-defined set of functionality. Components do not share state with other components, can be used unmodified in different contexts, and communicate only through their exported interfaces.

Typically, components have the following traits:
- Can be used by different applications written in a variety of programming languages
- Do not have source code or binary dependencies with other components
- Communicate with each other by exchanging messages
- Can be distributed over the network

The difference between components and class objects in Object-Oriented Programming (OOP) languages is two-dimensional. On the one hand, OOP encourages classes and their objects to be used for modeling real-world entities, taxonomies, and the interaction between them whereas component-oriented design just aims to group functionality and is indifferent to taxonomical disseminations. On the other hand, objects usually tend to depend and share state with other objects whereas, by definition, components are completely isolated and self-contained.

Thus, components are considered a higher level abstraction than objects. Essentially, components can be modeled and implemented by OOP objects. However, that does not mean that all objects fulfill the requirements of components.

3.2 Existing Component Frameworks

The Common Request Broker Architecture (CORBA) [19] is a standard that defines a set of specifications for creating and using software components that can be distributed over the network. CORBA uses an Interface Definition Language (IDL) to specify the exported interface of its components. The IDL interface is subsequently mapped to specific languages that implement the CORBA standard. That way, components can be created in any of the supported languages and interoperate seamlessly with each other. The IDL meta-compiler is used to generate automatically the stubs and skeletons that are essential for encoding and decoding respectively the objects that participate in a method invocation operation. In addition to the component infrastructure, the standard also defines a large set of services that can be used by CORBA-enabled applications.

Java Remote Method Invocation (Java RMI) [24] is a mechanism for enabling the invocation of methods that belong to distributed Java objects. Java RMI initially supported only objects written in the Java programming language. However, subsequent releases enabled the interoperation with objects written in other languages as well. Java RMI does not utilize an IDL meta-compiler and achieves the automatic generation of skeletons and stubs through the extensive introspection data that are built into the language. This mechanism along with a set of rules that a java class

must adhere to (e.g. Java Beans [25] and OSGi Bundles [21],) essentially define a complete component framework.

While the aforementioned systems are more focused on providing a middleware solution for distributed systems, Microsoft's Component Object Model (COM) [6] and Mozilla Foundation's Cross-platform Component Object Model (XPCOM) [27] target mainly desktop applications that run on a single machine. Both COM and XPCOM derive their architecture from the CORBA standard, implementing a subset of its specifications. Therefore, those systems also use an IDL meta-compiler to achieve the automatic generation of object stubs and skeletons and can be used for writing components in different programming languages.

3.3 Proposed Component Framework

Sparrow is a large scale system. However, it can be easily decomposed into distinct functional units with clearly defined roles and responsibilities. Consequently, it was decided to model these units using the component abstraction.

After reviewing the major component frameworks that were available at the time of designing the IDE (see section 3.2,) it became clear that none of them would be suitable for the intended purposes. On one hand, it was not possible to use the component frameworks that target the Java programming language, since Sparrow is implemented in C++. On the other hand, the two most robust and widely used C++ frameworks, COM and XPCOM, were unable to elegantly satisfy the set of requirements that were set for Sparrow's component system. The rationale is presented throughout the following sections.

3.3.1 Primary Requirements

The requirements that drove the design and implementation of Sparrow's component infrastructure are the following:

- Ease of use

16

- Runtime and memory efficiency
- Ability to load and unload components at runtime
- Ability to extend or reduce a component's exported interface at runtime
- Ability to make components automatically visible to other programming languages
- Ability to program components in other programming languages

The first two goals for the component architecture are apparent. Any software subsystem should be efficient and easy to use. Even much so when it constitutes a central part of the program and is intended to be used extensively in it.

Loading and unloading components at runtime was essential for implementing runtime adaptation and dynamic extensibility. Enabling the components that comprise the interface and the functionality of the IDE to be loaded and unloaded while it is running allows for many fundamental and diverse variations.

Another goal for the subsystem is the runtime extension or reduction of a component's interface by adding or removing methods. The author of a specific component may choose to enable or disable some of its functionality at some point in time, depending on the changes of the environment under which the IDE is running. E.g., an online-poker-game component may choose to offer a "Bet" method as long as the user's credit card has not reached its limit and disable it if it has. A discussion of whether poker functionality would be useful for an IDE, however, is beyond the scope of this Thesis.

Making components automatically visible to other programming languages and especially to the Delta programming language is very important. Forcing the programmer of a component to make it available explicitly to all the supported languages via wrappers is tedious and error prone. Additionally, if a new extension language is added at a latter time to Sparrow, all the available component wrappers would need to be updated.

Apart from allowing extension languages to interact with the existing components, a complete extension language mechanism should allow the coding of new components in the extension language itself.

3.3.2 Technical Overview

Essentially, a Sparrow component denotes a modular software unit that provides encapsulated reusable functionality to the IDE and its extensions. Sparrow components are typically, but not always, visual in nature and have the following characteristics:

- They can communicate with other components only through synchronous exchange of encoded messages
- They can be loaded and unloaded at runtime as their code is compiled as a Dynamic Link Library (DLL) [12] or as Delta bytecode
- They can form component hierarchies
- They can emit signals that trigger the invocation of slot methods that are contained inside the components that are interested in it
- They can inherit functionality from other components
- They encapsulate a property map, a hierarchical structure of user commands, and versioning metadata

Sparrow components are completely isolated from each other. Even if they use other components, they do not have hard coded dependencies with them. This is achieved by allowing the components to invoke the methods of other components only by sending messages. This isolation is further emphasized by the fact that a component is compiled into a DLL or Delta byte code and is loaded at runtime by the system whenever it is needed.

Components are designed to support the construction of containment hierarchies. This was done to ease the management of visual components and provide an intuitive model for combining different components in order to produce Sparrow's Graphical

User Interface (GUI). As is the case with the widgets metaphor in most GUI libraries, Sparrow's components can have another component as a parent and any number of child components. That way, components that belong to the same hierarchical structure can be notified from their parent or children about various events (see section 3.3.3.3) and can be automatically destroyed when their parent component is destroyed.

Signals and Slots are a flexible and intuitive mechanism for notifying interested clients about the occurrence of a specific event. It is also particularly useful in implementing the ubiquitous Observer pattern [8]. Sparrow's component infrastructure enables components to emit any number of signals that are differentiated by a unique identifier. At the other end, any component that is interested in a specific signal can register, at runtime, an exported method that is part of its interface as a slot to that signal. The slots of all the interested components are called as soon as the signal is emitted.

Although inheritance mechanisms are not common in other component frameworks, Sparrow's components are able to inherit and reuse functionality from other components. However, the component inheritance does not denote an *isA* relationship between the participating components. In fact an *isA* relationship, as discussed in section 3.1, is not meaningful or desired in component-oriented architectures. Therefore the component inheritance in Sparrow provides a way to reuse the functionality of specific components in case different components exhibit similar properties and functionality. Sparrow makes extensive use of component inheritance.

Sparrow's components incorporate a property map that associates a property with a specific value. Those property values are visible to the user, who is able to change their value and modify the behavior of a component. The property mechanism provides yet another means for the user to change the behavior of the IDE at runtime.

Additionally, a hierarchical structure of user commands is included in each component. A user command is simply a named callback to an exported method that is part of a component's interface. As soon as a component is loaded, its user commands are merged with the existing commands and appear as options to the IDE's

GUI, available for its user to invoke them. The author of a component can decide whether a user command will appear to the IDE's menu-bar, its toolbar, or in a context menu. Despite the close connection between user commands and some visual elements of the IDE's GUI, it is essential to point out that user commands by themselves do not have any connection with these visual elements nor depend on the specific GUI library that is used in Sparrow. They merely contain platform independent data that are interpreted by their container and are subsequently realized as visual elements.

Lastly, each component contains a set of metadata that provide the human friendly name of the component, a short description of its functionality and role in the system, the identity of its author, and its version. These data on one hand are used for the self-documentation of the system (see section 5) and on the other hand provide versioning information to the user of a component; and someone to blame when the component fails to work as expected.

3.3.3 Implementation Details

The usage of Sparrow components closely resembles the usage of COM and XPCOM component models. However, an Interface Definition Language (IDL) compiler is not utilized for the specification of the interface of a Sparrow component. Sparrow components do not implement a static interface; they essentially construct their interface at runtime by exposing a set of native methods. Thus, it can be inferred that Sparrow components implement a *concept* rather than an *interface*. Therefore, in Sparrow, a component's *concept* essentially constitutes that component's API.

Sparrow's component system makes a distinction between component classes and component instances. The relationship between classes and instances has solely runtime semantics. That means that a component instance that belongs to a specific class maintains a reference to a structure that is constructed at runtime and plays the role of its class. Thus, a Sparrow component, at a lower level, is essentially the runtime model of a specific component instance and its corresponding component class. Multiple instances of the same class share the same component class object.

As seen in Figure 5, a component class contains the following: (a) a set of all the available functions that a component exports, (b) a list of the slots that are triggered whenever a signal is emitted, (c) a list of signals that the component emits, (d) a set of properties that affect the functionality of the component, (e) a hierarchical structure of user commands, (f) a list of the component instances that are associated with the specific component class, (g) versioning metadata, (h) a reference to the base component class of the current class, and (i) a list of references to the component classes that derive from the current class.

A component instance (Figure 5), contains the following elements: (a) a monotonically increasing serial number that identifies different instances, (b) a reference to the parent component instance, (c) a list of references to the child instances, (d) a set of properties that affect the functionality of the specific instance, and (d) a reference to the component class in which the current instance belongs.

A component function (Figure 5), contains four elements: (a) the return type of the current function as string, (b) a vector of strings that represent the types of the function's arguments, (c) a documentation string that describes the function, and (d) a native function pointer to the actual low level function that is called whenever the component function is invoked.
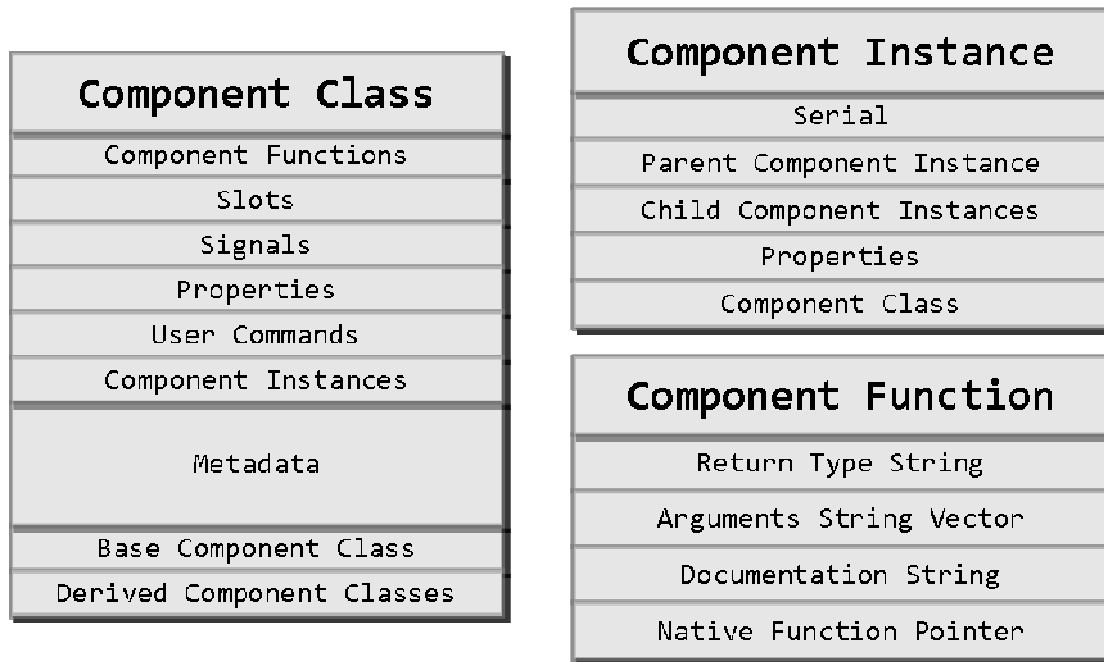
Figure 5 - The basic building blocks of a Sparrow Component

Most of the contained elements of the basic building blocks of a Sparrow component (Component Class, Component Instance, and Component Function) are described in the previous section. Nonetheless, some of the elements and their purpose in the Sparrow component model require some clarification.

The monotonically increasing serial number that is part of an instance is essential for referencing a specific component since it is merely the runtime model of a component instance and its corresponding class. Thus, Sparrow components can be uniquely referenced by the pair *(component class, serial number)* which represents a *handle* to a single component. Whenever a client wants to communicate with a component, it just needs to obtain or construct a handle to it. Using handles as a means to reference components has many advantages over using raw pointers. Dangling handle references that may be kept by a client do absolutely no harm to the system when accessed, whereas if raw pointers were used, the whole system would crash in case they were dereferenced. Additionally, handles have the advantage of being capable of referencing components that may reside in a different process or in a different language or both.

Property maps are present in both classes and instances. This redundancy enables the user to either affect the functionality of a specific component instance or the functionality of all the instances that belong to the same class. E.g., a source editor component will usually have class-wide properties for controlling the look of highlighted text, so that all the editors will have consistent appearance. However, it is preferred to have a "toggle highlight" property only as an instance property, so that changes to its value will affect just a specific editor.

The component function structure holds the types of the target function's arguments and return value. These records are used for supplying introspection information to the component system and are essential for bridging the native C++ types to the types of other programming languages (e.g. Delta) so that inter-language component communication can be realized. Again, the documentation string is used for the self-documentation of the IDE. The structure also holds a pointer to the actual native function that is finally invoked. The native function though, is actually the skeleton for the real component function that is invoked. The responsibilities of the skeleton are the following:

- Decode from the supplied message the arguments of the real function
- Invoke the real function which may be a Delta or C++ function supplying the decoded arguments
- Encode the return value of the function as a message

Component functions can be either *static* or *member*. The difference between them is that member functions must be called inside the context of a specific component instance, whereas static functions do not require any instance. Thus, a Sparrow component, in addition to being the runtime model of a class and an instance, can also be realized simply by a component class object that holds exclusively static functions.

Sparrow's basic component framework is complemented by the Component Registry, the Component Loader, and the Component Factory. The Component Registry is responsible for holding and managing all the available component classes in the system. Its main role is actually being the entry point for accessing the components. The Component Loader is able to load Delta and C++ (DLL) components from the

disk on demand (see section 3.3.3.3) and register them to the Component Registry so that they can be used by the IDE. Lastly, the Component Factory is responsible for creating and initializing component instances of a given component class. It handles the creation of each instance by querying its corresponding class from the Component Registry and retrieving its *constructor* function; which is included in all non-static Component Class objects. The basic architecture of the Sparrow's component subsystem can be seen in Figure 6. It is worth noting that the aforementioned elements are realized as Singletons [8].
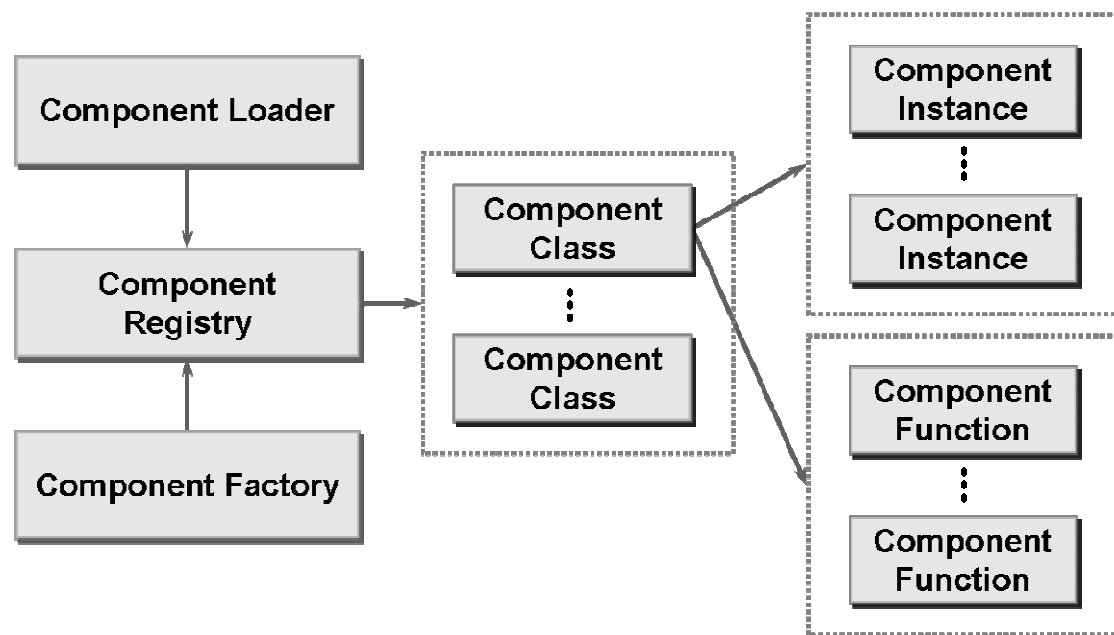
Figure 6 - Runtime dependencies between the basic building blocks of Sparrow's component framework

3.3.3.1 Inheritance

As mentioned in section 3.3.2, component inheritance in Sparrow provides the means to reuse functionality between components. A component class object can inherit another class object. By doing so, it automatically inherits (a) the component functions, (b) the slots, (c) the properties, and (d) the user commands of its base class.

The functions and slots are not copied to the derived class object and any base class function can be overridden. Therefore, the calling mechanics of the functions in a

component inheritance hierarchy are implemented by the message dispatching algorithm. Specifically, the simple lookup algorithm used for retrieving a function, can be seen in Figure 7. Section 3.3.3.2, additionally, outlines an example invocation of a function in an inheritance hierarchy.

```
lookup (component, func) {
      if func is found in component.functions then
            return component.functions[func]
      else if component has base class then
            return lookup(component.baseClass, func)
      else
            return nil
}
```

Figure 7 - The recursive lookup algorithm for component functions

The properties of a base class are copied verbatim to the derived class object. Similarly, the user commands are also copied, but with a small modification: the callback part of the user command, which typically comprises of the pair *(component class, component function)*, is altered to reference the most derived class. Obviously, the component function part remains unchanged. That way the user commands point always to the correct function in case one is overridden. Lastly, because of the fact that all the user commands and properties are added to the component class object at runtime, it is essential for the system to synchronize their additions and removals in the whole inheritance hierarchy. This is the main reason why the component class objects maintain references to their base and derived classes (see Figure 8.)
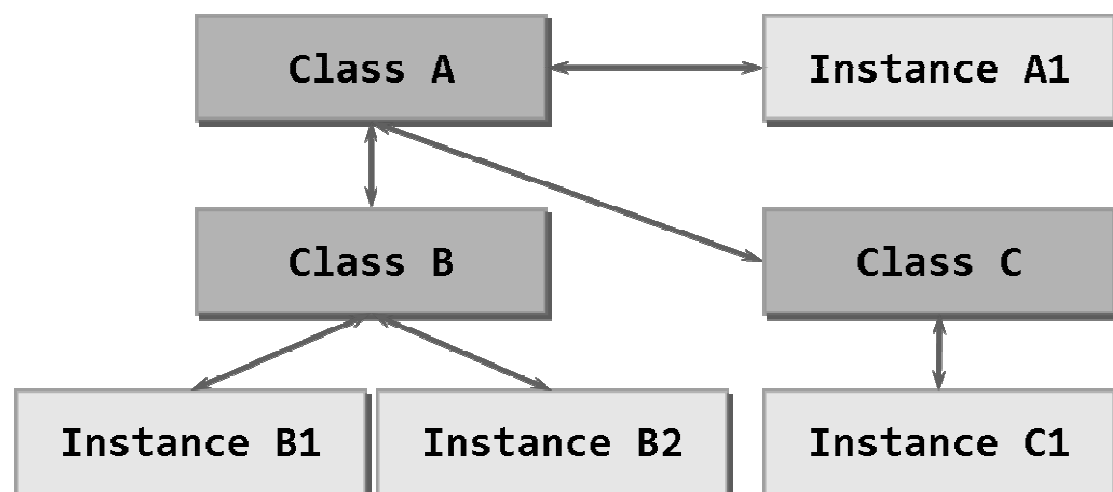
Figure 8 - Runtime model of classes and instances in a scenario that utilizes component inheritance

3.3.3.2 Invocations

Sparrow components are only allowed to interact with other components through message passing. Thus, each exchanged message must contain (a) a reference to the destination object, (b) the name of object's method that needs to be called, and (c) a list of values that are used as the function's arguments. Provided that a function can have any type and number of arguments, the value list that is embedded inside the message is encoded as a data buffer and is decoded at the destination function's skeleton. Messages between components are exchanged only within the IDE's process (i.e. within the same address space.) Thus, whenever component *A* wants to call a specific function that belongs to component *B*, it simply sends a message to *B* containing the name of the method that it wants to invoke along with an encoded argument list that is passed to that function. Subsequently, component *B* returns the result of the invocation, i.e. the return value of the function, as an encoded buffer.

The communication between the components is arbitrated by an entity called Message Router. Message Router is responsible for receiving requests for component invocations and dispatching them to the appropriate component function. The message exchange process through the Message Router is synchronous. That way, actual component functions can have "regular" function semantics: they can return any value that the component author wishes, and they can conveniently throw exceptions. This is not the case in COM or XPCOM in which all the functions return a predefined status value that indicates whether the invocation was successful, exceptions are prohibited by law, and potential return values are typically passed as reference-to-lvalue arguments. Allowing the component functions to have "regular" function semantics was deemed as important for achieving the "ease of use" goal for the component subsystem.
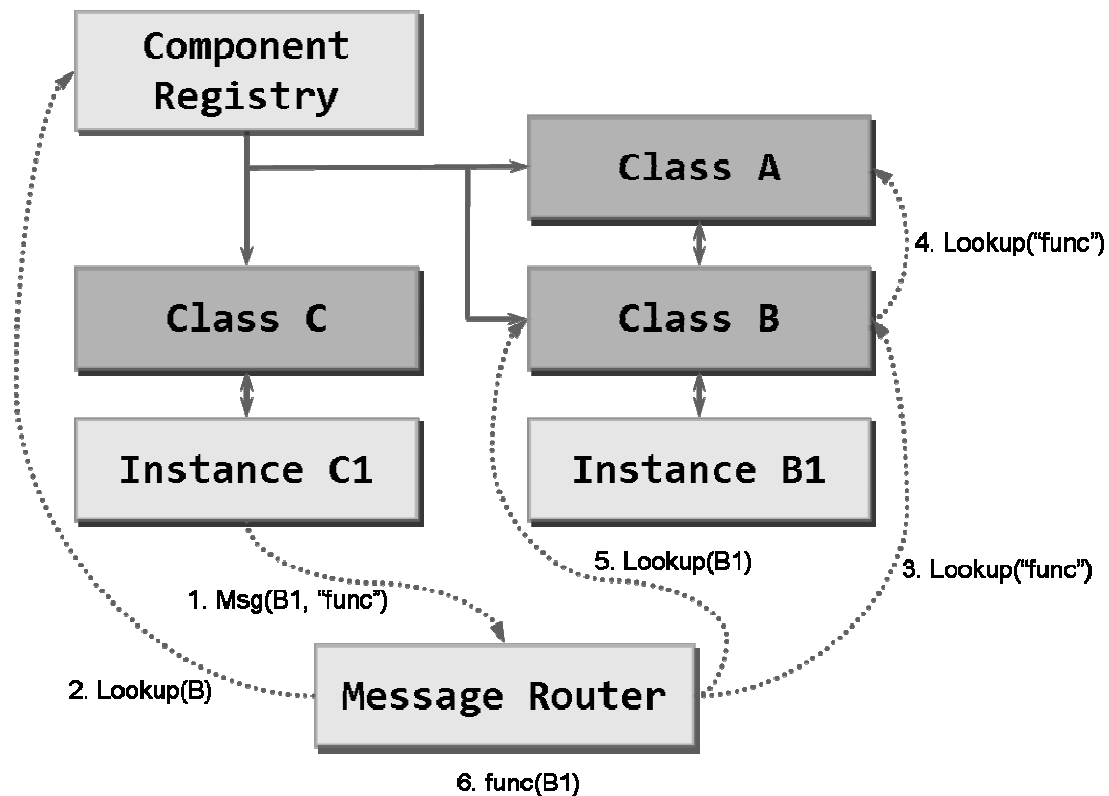
Figure 9 - Inter-component communication through message passing

The message dispatching process can be seen clearly in the example shown in Figure 9; where the component *C* wants to call the function *"func"* of component *B*. In that scenario, the following steps are performed:

1. Message Router (MR) receives a request to call function "func" of instance B1
2. Through the Component Registry MR retrieves "Class B" entry
3. MR queries "Class B" entry to find "func"
4. In an unfortunate turn of events, "func" is not contained in "Class B" so the lookup algorithm searches for "func" recursively in its base classes and locates it in "Class A"
5. MR searches and retrieves instance B1
6. MR invokes "func" in the context of component instance B1

3.3.3.3 Notifications

Sparrow's component framework implements an internal notification mechanism that enables components and other subsystems to be notified about changes in the state of

the component infrastructure. Three notification contexts are distinguished: (a) the context of the current instance in which an event takes place, (b) the context of the child instance (which is part of a containment hierarchy) in which an instance learns of an event that happened in one of its children, and (c) the global context in which events from all the components are mirrored.

In Table 2 all the internal notifications that are supported by the component system can be seen in the leftmost column. The other columns, that reference the different notification contexts, indicate whether the notification is supported in a specific context.

Table 2 - Standard internal component notification**s**

Notification description	Current Instance	Child Instance	Global
Requested access to nonexistent class	✖	✖	✔
Registered a component class	✖	✖	✔
Unregistered a component class	✖	✖	✔
Created first instance of class	✖	✖	✔
Destroyed last instance of class	✖	✖	✔
Created instance	✖	✖	✔
Destroyed instance	✖	✖	✔
Added a component instance to a hierarchy	✔	✔	✔
Removed a component instance from the hierarchy	✔	✔	✔
Destroyed a component instance	✔	✔	✔
Added a component instance to a hierarchy as first of its class	✔	✔	✖
Removed a component instance from a hierarchy as last of its class	✔	✔	✖
Destroyed a component instance as last of its class	✔	✔	✖
Component instance focused	✔	✔	✔
Applied changed properties	✔	✔	✔
Added a property	✔	✔	✔
Removed a property	✔	✔	✔
Added a function	✔	✔	✔
Removed a function	✔	✔	✔
Added a user command	✔	✔	✔
Removed a user command	✔	✔	✔
Merged user commands	✔	✔	✔
Unmerged user commands	✔	✔	✔
Added signal	✔	✔	✔
Removed signal	✔	✔	✔

One issue worth discussing here is the "Requested access to nonexistent class" notification. This specific notification is triggered whenever a component class is requested from the Component Registry (e.g. as a result of a call to a specific component function, or a Component Factory construction operation) and is not available among the registered classes. On the other hand, the Component Loader "listens" for this specific event and when it is triggered it tries to load the missing component from the disk. Thus, only when the Component Loader fails to retrieve the missing component, the initial operation, that triggered the signal in the first place, fails.

3.3.3.4 Component Specification Language

Having the components being constructed entirely at runtime introduces many difficulties when authoring their code. The programmer is required to write boilerplate code for exporting each function that is part of a component's interface, for registering user commands, for registering signals and slots, and even for creating the skeletons for all the functions. This is not only extremely tedious but also highly error-prone.

Other component systems, such as COM and XPCOM, mitigate the aforementioned problem by having static interfaces[1] and by utilizing their IDL meta-compiler that generates most of the boilerplate code. However, the use of an IDL meta-compiler was not a viable option for Sparrow. Having an immutable interface for each of the components, on one hand would limit their flexibility, and on the other hand would necessitate the implementation of an IDL for each of the supported extension languages (i.e., Delta.) Also, the fact that such an approach would not allow the extensibility of a component's exported API at runtime, further justifies the argument against the deployment of an IDL meta-compiler in Sparrow.

---

[1] Actually, COM and XPCOM support the construction of an interface at runtime, but in this case they do not provide any means to eliminate boilerplate code

Sparrow automates the generation of all the boilerplate code by introducing a Domain Specific Language (DSL) for authoring components. The term DSL, generally, describes a micro language that provides an intuitive syntax and semantics for solving problems that reside in a very specific and constrained domain. Sparrow's Component Description DSL is implemented using a mixture of C++ templates and preprocessor macros (utilizing the boost preprocessor library.) To paraphrase a famous saying: "A snippet of code is worth a thousand and twenty four words," hence, the basic aspects and usage of Sparrow's component description DSL are shown in Figure 10.

The aforementioned DSL was also extended to allow an intuitive syntax for calling component functions. That aspect of the DSL, effectively, automates the creation of function stubs. This is shown in Figure 11.

```
// File: HelloWorld.h

class HelloWorld : public Component {
      DECLARE_COMPONENT(HelloWorld);

public:
      DECLARE_EXPORTED_MEMBER(void, SetValue, (const string& value));
      DECLARE_EXPORTED_MEMBER_(const string&, GetValue, (void),
            _("Retrieves the value"));
      DECLARE_EXPORTED_MEMBER(void, Print, (void));
      DECLARE_EXPORTED_STATIC(void, PrintValue, (const string& value));
      DECLARE_EXPORTED_MEMBER(void, Show, (void));
      DECLARE_EXPORTED_MEMBER(void, SlotStringShown, (void));

private:
      string m_value;
};
```

```
// File: HelloWorld.cpp

COMPONENT_METADATA(
      HelloWorld,                                  // class
      "HelloWorld",                                // name
      "Says hello world with style",               // description
      "Yannis Georgalis <jgeorgal@ics.forth.gr>",  // author
      "1.0"                                        // version
);
IMPLEMENT_COMPONENT(HelloWorld);

COMPONENT_SET_PROPERTIES_FUNCTION(HelloWorld, table)
      { /* We do not need any properties */ }

EXPORTED_MEMBER(HelloWorld, void, SetValue, (const string& value))
      { m_value = value; }

EXPORTED_MEMBER(HelloWorld, const string&, GetValue, (void))
      { return m_value; }

EXPORTED_MEMBER(HelloWorld, void, Print, (void))
      { PrintValue(m_value); }

EXPORTED_STATIC(HelloWorld, void, PrintValue, (const string& value))
      { cout << "Hello world, " << value << endl; }

EXPORTED_SIGNAL(HelloWorld, StringShown, (const string& str));

EXPORTED_CMD_MEMBER(HelloWorld, Show, _("/View/Show"), MT_MAIN)
{
      this->Print();
      sigStringShown(m_value);
}
EXPORTED_SLOT_MEMBER(HelloWorld, void, SlotStringShown, (), "StringShown")
{
      cout << "Signal StringShown TRIGGERED" << endl;
}
```

Figure 10 - A Sparrow component implementation in C++


```
// File: HelloWorldCaller.cpp

Component* component = ComponentFactory::Instance().Create("HelloWorld");

Call<void (string)>(component, "SetValue") func;
func("Innit?");

cout << Call<string ()>(component, "GetValue")() << endl; // Prints: Innit?

Call<void ()>(component, "Print")(); // Prints: Hello world, Innit?

Call<void (string)>("HelloWorld", "PrintValue") staticFunc;
staticFunc("hey?"); // Prints: Hello world, hey?

DCall<void>(component, "SetValue")(string("said the component"));
const string val = DCall<string>(component, "GetValue")();

// Prints: Hello world, said the component
//
DCall<void>("HelloWorld", "PrintValue")(val);
```

Figure 11 - Constructing a component instance and calling its methods

There are a few issues worth noting in the above examples:

- Component function declarations are very similar to the declarations of C++ class functions and they also have variants for supporting a documentation string to be attached to them

- The declared component functions are also legal C++ functions, and can be called as such

- The part of the DSL that automates the creation of stubs has two variants: the "Call", and the "DCall". The difference is that while the Call requires the provision of the full signature of a component function, the DCall only needs its return type. However, DCall has the disadvantage of not knowing how to convert the types of its arguments if they are different from the component function's argument types

- A slot to a specific signal can take fewer arguments than the signal; however, it cannot change their order. This functionality closely resembles Qt library's [28] slots

Using the DSL, the amount of the code for exporting the interface of a component is in fact less than that required by COM or XPCOM; apart from the IDL description, which is quite verbose, the programmer is also required to derive from the generated interface in order to implement the component's functionality. Also, since the DSL provides facilities for exporting user commands and describing signals and slots, it further reduces the amount of the required boilerplate code.

3.4 Extending Components

The creation of components, through the Component Factory, and the invocation of component functions, through message passing, do not impose any dependencies to concrete implementations. All the dependencies between implementations are implicit. That allows replacement or augmentation of components at runtime (horizontal extensibility). Actually, a component can be fully replaced by any other implementation as long as it (a) exports the same API or a superset of it, (b) emits the same signals or a superset of them, and (c) the functions and the signals, that are

32

replacing the originals, have similar semantic behavior in respect to the caller's assumptions.

Vertical extensibility through the component system is straightforward. New components can be registered and deployed at any time. Regardless of the programming language they were written in, they can use any of the other available components without any restriction.

As mentioned above, Sparrow's component framework, mainly due to its runtime nature, provides facilities to export components to different programming languages. Typical approaches for achieving this – apart from porting the whole component framework, which is CORBA's approach – utilize a separate tool that automates the process of inter-language interface exporting by generating wrapper code for the target language. Meta-compilers, such as SWIG [3] generate wrappers by parsing the source language's interface code. Sparrow's architecture, on the other hand, enables third-party languages to invoke any component function by implementing and exporting a library to the target language (e.g. Delta) that is able to perform the following tasks: (a) export the component construction and query mechanism, (b) encode and decode the exchanged messages, and (c) perform conversions between the types that take part in an invocation operation (i.e. argument types and the return type). A language that provides the means to implement these functions is capable of calling and interacting with any component that is available to Sparrow.

Other than the one-way communication support, the framework's architecture also facilitates the construction of components in other languages, in order to enable the extendibility of the IDE through them. Thus, language developers can extend the component infrastructure with a proxy that: (a) manages the creation and destruction of components that are built in the other language, and (b) dispatches the component calls that are directed to the managed components. The proxy, along with the component exporting mechanism described above, effectively completes the requirements of a Sparrow extension mechanism through third party languages.

The Delta Extensibility Layer [5] was implemented by utilizing these facilities. That two-way communication appoints Delta as an equal to C++ for extending the IDE's

functionality. However, since Delta lacks an implementation of a Graphical User Interface (GUI) extension library, it cannot be used for the implementation of visual components.

3.5 Global and Local Undo / Redo

Another goal for Sparrow was the provision of an undo mechanism that is capable of undoing and redoing the effects – visual or not – of any operation that changes the state of the IDE. Sparrow achieves such goal by providing an unobtrusive undo subsystem, orthogonal to the component infrastructure, that is able to record and replay inter-component undo invocations.

The main module of the undo subsystem is the Undo Manager. The Undo Manager offers an interface for components to register messages – as if they were calling a certain component function – that, when dispatched, have as an effect the cancellation of the current operation. Certain problems emerge, however, when, inside a single component invocation, multiple component functions in the call stack attempt to register their undo message.

In the case where function F invokes function G (F → G) and F', G' are the reverse functions for F and G respectively, the registration of both the undo calls F' and G' may introduce problems when F' also cancels the effects of G. The main assumption made in Sparrow's undo subsystem is that this will always be the case. That is whenever F → G then F' should *always* imply G'; so the registration of G' by G will be discarded. In case, however, F' is not provided then G' can serve as a reverse function for both F and G. That is true for any invocation depth (F → G → H → …) Additionally, when F → G and F → H (F→ G, H) and F' is not provided, then essentially the linear combination of G' and H' can serve as reverse calls for F. Again, that is true for any invocation breadth (F → G, H,…)

Hence, Sparrow's undo subsystem automatically enforces the aforementioned assumptions in order to provide an efficient – for the component programmer - undo/redo mechanism. By making the provision of an F' for every function F

optional, the undo subsystem successfully minimizes the amount of code that is needed for supporting universal undo/redo functionality in Sparrow, while maximizing reusability. The code needed for registering an action to the Undo Manager can be seen in Figure 12.

The multilevel characterization of the undo subsystem is attributed to its ability to maintain multiple undo queues (in addition to the global one) for every component that initiates a specific invocation. Thus, while a global undo operation cancels the effects of the latest invocation, an undo operation for a component cancels the effects of that specific component's latest invocation. This functionality is very useful in a dynamic system, like Sparrow, where the addition of a faulty component can leave the system in an inconsistent state. Using the undo subsystem, an extension language proxy can cancel the changes that are imposed by a faulty script that exits prematurely with a runtime error.

```
// File: FakeWindow.h

class FakeWindow : public Component {
      DECLARE_COMPONENT(FakeWindow);

public:
      DECLARE_EXPORTED_MEMBER_(void, SetTitle, (const string& title),
            _("Sets the title of the window"));

      DECLARE_EXPORTED_MEMBER_(const string&, GetTitle, (void),
            _("Retrieves the title of the window"));

      DECLARE_EXPORTED_MEMBER(void, ClearTitle, (void));

private:
      string m_title;
};
```

```
// File: FakeWindow.cpp

COMPONENT_METADATA(
      FakeWindow,                              // class
      "Fake Window",                           // name
      "Represents a window with undoable actions",  // description
      "Yannis Georgalis <jgeorgal@ics.forth.gr>",   // author
      "1.0"                                    // version
);
IMPLEMENT_COMPONENT(FakeWindow);

COMPONENT_SET_PROPERTIES_FUNCTION(FakeWindow, table)
      { /* We do not need any properties */ }

EXPORTED_MEMBER(FakeWindow, void, SetTitle, (const string& title))
{
      Undo<void (string)>(this, "SetTitle")(m_title);
      m_title = title;
}

EXPORTED_MEMBER(FakeWindow, void, ClearTitle, (void))
{
      Undo<void (string)>(this, "SetTitle")(m_title);
      m_title.clear();
}

EXPORTED_MEMBER(FakeWindow, const string&, GetTitle, (void))
      { return m_title; }
```

Figure 12 - A C++ Sparrow component that supports Undo/Redo

4. Remote Component Deployment

The "meta" dimension of Sparrow is mainly enforced by creating or extending existing components in order to assemble a tool-chain that serves the development needs of specific problem domains. Nonetheless, Sparrow's component framework, while being easy to extend and use, can be too intrusive for incorporating into existing software systems. Existing software systems may employ their own set of libraries, frameworks, or even use a completely different programming language. Reorganizing or rewriting the code of these systems for making them suitable to be hosted under the Sparrow tool-chain may be prohibitively expensive.

To overcome the aforementioned barriers, Sparrow provides a method to deploy the IDE from other programs and exploit its functionality without requiring major changes in their infrastructure. In the following sections, the design, implementation, and functionality of Sparrow's remote deployment subsystem will be presented.

4.1 Technical Approach

The ability to interact with the available components from other processes was deemed essential for the realization of the inter-process deployment goal. Enabling another process to invoke all the functions that are exported by Sparrow – that runs in its own process – allows for the complete remote manipulation of the IDE.

By definition, the remote deployment of the IDE from third-party applications requires an Inter-Process Communication (IPC) mechanism. When designing the deployment mechanism of Sparrow, it was considered useful that the inter-process communication be implemented over a network protocol. This would enable the programs that deploy the IDE to run on a remote computer – different from the one that would run Sparrow. Thus, the remote deployment mechanism was implemented over a lightweight TCP/IP protocol.

Such implementation makes all the available components accessible to any remote process that can establish a connection to Sparrow's process. However, it was also observed that for most remote deployment needs just a small subset of Sparrow's functionality would be enough. For that reason, a library that simplifies the tasks that were considered essential for the deployment of the IDE was also implemented. This library essentially offers higher level abstractions for managing Sparrow as it encapsulates higher-level tasks that require more than one component invocation.

4.2 Implementation Details

The key elements of the remote deployment subsystem are the Message Router Client (MRC) and the Message Router Server (MRS). MRS runs in the same process as the IDE and encompasses the functionality of a TCP/IP server that listens to a predefined port. All established connections to Sparrow are managed by MRS which is also responsible for dispatching the received messages through the Message Router – the central point of the inter-component communication. On the other hand, the MRC is responsible for connecting to the MRS and forwarding all the requests submitted by its clients.

Because all the components in Sparrow interact with each other through messages, no conversions need take place during the life time of a remote call. The initial message submitted by the remote caller is a well formed invocation message for a specific component that resides in the IDE process. The only job of MRC is, thus, to forward the message to the MRS and MRS in its turn just forwards it to the Message Router.
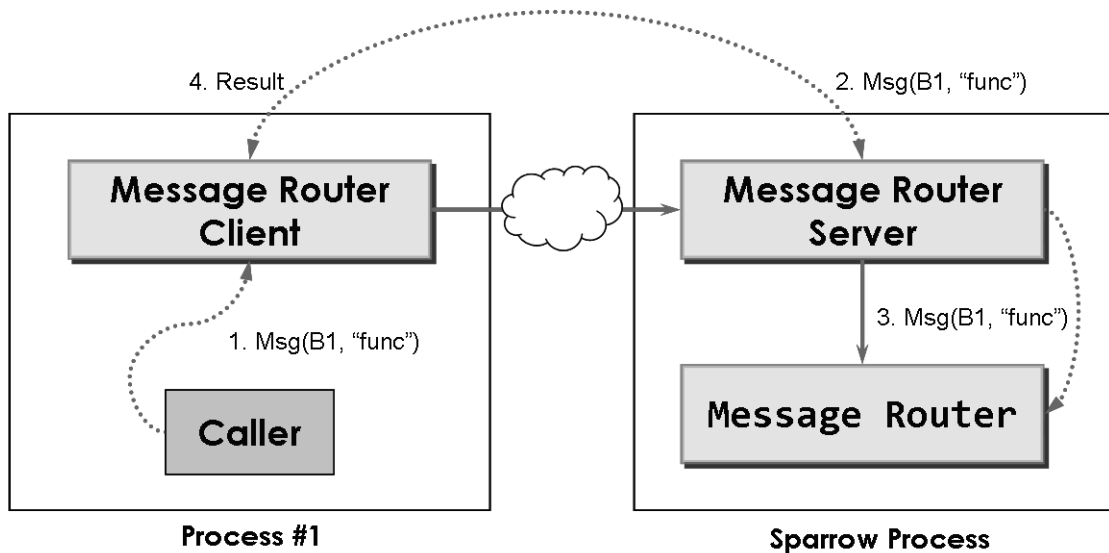
Figure 13 - Remote component invocation

The remote invocation process is explained in Figure 13, where the *Caller* wants to call the function *"func"* of component *B*. In that scenario, the following steps are performed:

1. Message Router Client (MRC) receives a request to call function "func" of instance B1

2. The message that encodes the request is forwarded unmodified to the Message Router Server (MRS)

3. MRS dispatches the message through the Message Router

4. MRS subsequently sends through the connection, on which the request arrived in the first place, the result of the invocation (i.e. whether it was successful or not) and the return value of the invocation

Another thing worth noting, concerning the implementation of the remote deployment subsystem, is that, while the TCP/IP server of MRS is running in its own thread inside the Sparrow process, the message dispatching process - through the Message Router - is actually executed from the main thread. That way, potential race conditions are eliminated and component programmers are not burdened with the unnecessary – in this context - overhead of multithreaded programming.

4.3 IDE Deployment API

As mentioned above, it was expected that the majority of applications that would like to deploy Sparrow would require only a small subset of its functionality. Thus, in order to make it easier for programmers, Sparrow offers an API that encapsulates the most common functionality. The set of functions that are available to applications that deploy the IDE appears in Table 3.

Table 3 - The exported Deployment API

Function	Description
void OpenWorkspace (string uri)	Opens the workspace that is referenced by the given URI
void CloseWorkspace (void)	Closes the current workspace
void NewWorkspace (void)	Creates a new workspace and makes it the current
void RenameWorkspace (string name)	Renames the current workspace to the given name
void AddProject (string uri)	Adds a project, referenced by the given URI, to the current workspace
void RemoveProject (string name)	Removes the project with the given name
void NewProject (string name)	Creates a new project with the given name
void RenameProject (string name, string newName)	Renames the project with the given name to the given new name
void AddFile (string projectName, string uri)	Adds the file, referenced by the given URI, to the given project
void RemoveFile (string projectName, string name)	Removes the file with the given name from the given project
void NewFile (string projectName, string name)	Creates a new file with the given name and adds it to the given project
void RenameFile (string projectName, string name, string newName)	Renames a file, which is contained inside the given project, with the given name to the given new name
void LoadProfile (string name)	Loads the profile with the given name

The deployment API has two end-points. On the client side, where the application that deploys Sparrow resides, the API is exported as a Dynamic Link Library (DLL). The same API is also mirrored at the server side, where the Sparrow process resides. In the Sparrow process, the deployment API is realized as a typical component - loaded on demand. Essentially, the client deployment API is a wrapper for the remote invocation of the "Deployment API" component. That way the deployment API itself maintains the "meta" attribute of the IDE, since in case Sparrow is deployed in a different

context, the API can be extended or replaced to reflect the specific needs of the problem domain like any other component.

4.4 Examples of Use

The deployment infrastructure of Sparrow enables a remote process to programmatically invoke the components that reside in the IDE using a similar method as the one used for inter-component communication. The generic method for calling any component through the facilities provided by the deployment library can be seen in Figure 14, while the use of the equivalent wrapper functions is displayed in Figure 15.

The wrapper functions' code is just shorthand for the invocations presented in Figure 14. They also serve as a means to minimize the compile-time dependencies and include files needed for the applications that deploy Sparrow.

```
// File: RemoteCaller.cpp

ext::DeploymentAPI::Initialize(_T("localhost"));

RCall<void (void)>("DeploymentAPI", "NewWorkspace")();

RCall<void (string)> openWs("DeploymentAPI", "OpenWorkspace");
openWs("C:\Etc\Passwd");

// Any component can be invoked remotely
RCall<void (string)>("Editor", "OpenFile")("C:\Etc\Passwd");

ext::DeploymentAPI::CleanUp();
```

Figure 14 - Generic remote invocation

```
// File: RemoteCallerDeploymentAPI.cpp

ext::DeploymentAPI::Initialize(_T("localhost"));

ext::DeploymentAPI::NewWorkspace();
ext::DeploymentAPI::OpenWorkspace("C:\Etc\Passwd");

ext::DeploymentAPI::CleanUp();
```

Figure 15 - Using the deployment API

5. Interactive Introspection

The primary roles of the interactive introspection tool are the documentation and debugging of the system. Interactive introspection serves as a tool to extract and display the documentation that describes the components of the IDE and their functions. Also, by being interactive, it allows the programmer to fiddle with the components that are currently active in the system and observe their behavior in real time.

Interactive introspection is mainly targeted at the component programmers that wish to extend or deploy Sparrow; it proved to be an indispensable tool during the development of Sparrow, as it made it possible to immediately test and observe the functionality of the components that were being developed.

In the following sections the design, implementation, and usage of Sparrow's interactive introspection will be presented.

5.1 Technical Approach

Interactive introspection is implemented in Sparrow as a component offering a graphical user interface. The said component, tentatively named Component Spy (CS,) extracts and displays all the built-in introspection data that are embedded inside components. Specifically, CS displays the following information for each component:
- Documentation
- Author's name and e-mail
- Version
- Base component class
- Derived component classes
- Properties
- Exported functions
    - Documentation
    - Return type

- o Argument types
- Signals
  - o Argument types
  - o The slots that are connected to it
- Created instances
  - o Serial number
  - o Child instances
  - o Properties

This kind of information is provided explicitly or implicitly by the programmer when constructing components - using the component description DSL (see section 3.3.) The method of information registration for components implemented in Delta is described in [5].

The provided information and the way it is presented offers programmers a comprehensive reference for invoking components, connecting slots to existing signals, and understanding the organization of Sparrow's architectural elements.
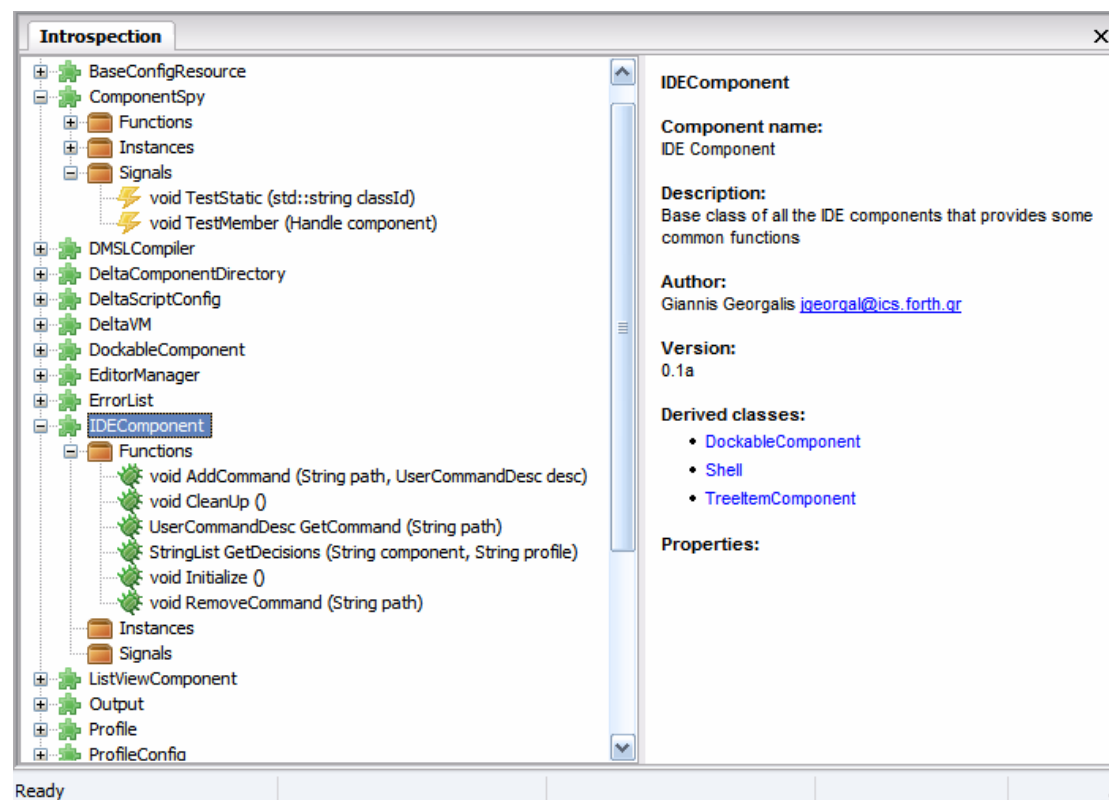
Figure 16 - Interactive component introspection interface

In addition to viewing the introspection data that are encapsulated in the components, users can perform the following actions:

- Unload a component
- Configure a component by changing its properties
- Invoke a component function
- Remove a component function
- Delete a component instance
- Remove a signal from the component

These actions were found to be very useful for debugging the system, or to test its behavior under corner cases. Unloading a component removes it from the memory and deletes all its instances. Removing a component function essentially makes it inaccessible from that point on. Correspondingly, removing a signal prevents the component from emitting it, and slots can no longer connect to it. Deleting a component instance forces the system to release its resources, remove any graphical elements that are associated with it, and delete all its contained child instances recursively. These destructive operations can be proved valuable in testing how components behave if one or more of the elements they depend on cease to exist. On the other hand, through Component Spy, the user can invoke any component function and see the effects of its invocation immediately.

5.2 Implementation Details

When Component Spy is instantiated, it queries all the components that are currently active and extracts all the relevant data. It subsequently builds a tree-view structure to organize the component data and their contained elements. After the successful construction of the tree-view interface, Component Spy registers itself as a listener to the Component Registry (see section 3.3) in order to monitor the component infrastructure for changes that affect its visualization structures. Specifically, Component Spy listens for all the global notifications (see section 3.3.3.3) and

updates the tree-view to reflect the current state of the components in order to maintain it synchronized. That way the visualized data are always consistent.

5.3 User Interface

The interface of Component Spy is comprised of a tree-view widget and a text-view widget. The text-view structure displays: (a) the components as top-level nodes, (b) the component instances as child nodes of the component nodes, and (c) the signals as child nodes of the component nodes. The component instances are organized in a sub-tree that exposes their containment hierarchy (see Figure 19.) The text-view structure, on the other hand, displays context sensitive information that depend on the selected tree-view item.

The text-view in Figure 17 provides additional information on the selected component, which is – in this case – Component Spy. Figure 18 displays the documentation of a function, whereas Figure 19 presents information on the selected instance. The icon that appears on the left of each component function is a subtle but bold reminder that all functions are potential bug-hives.
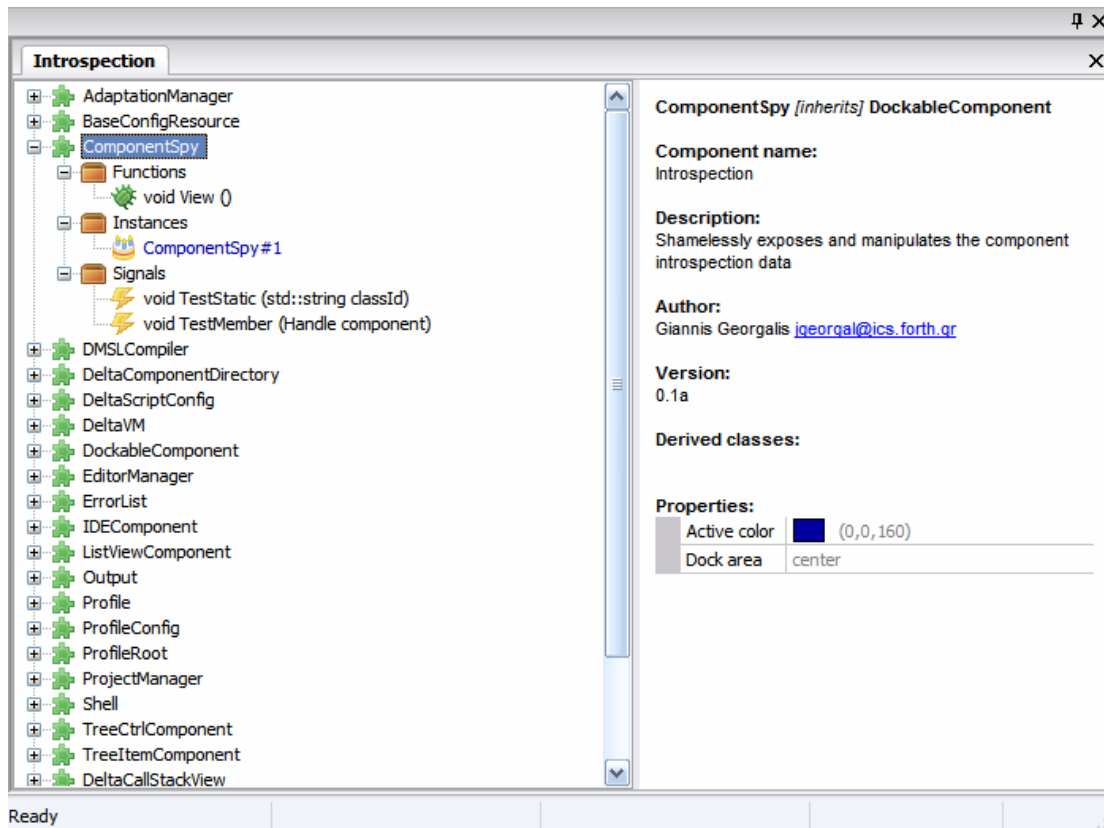
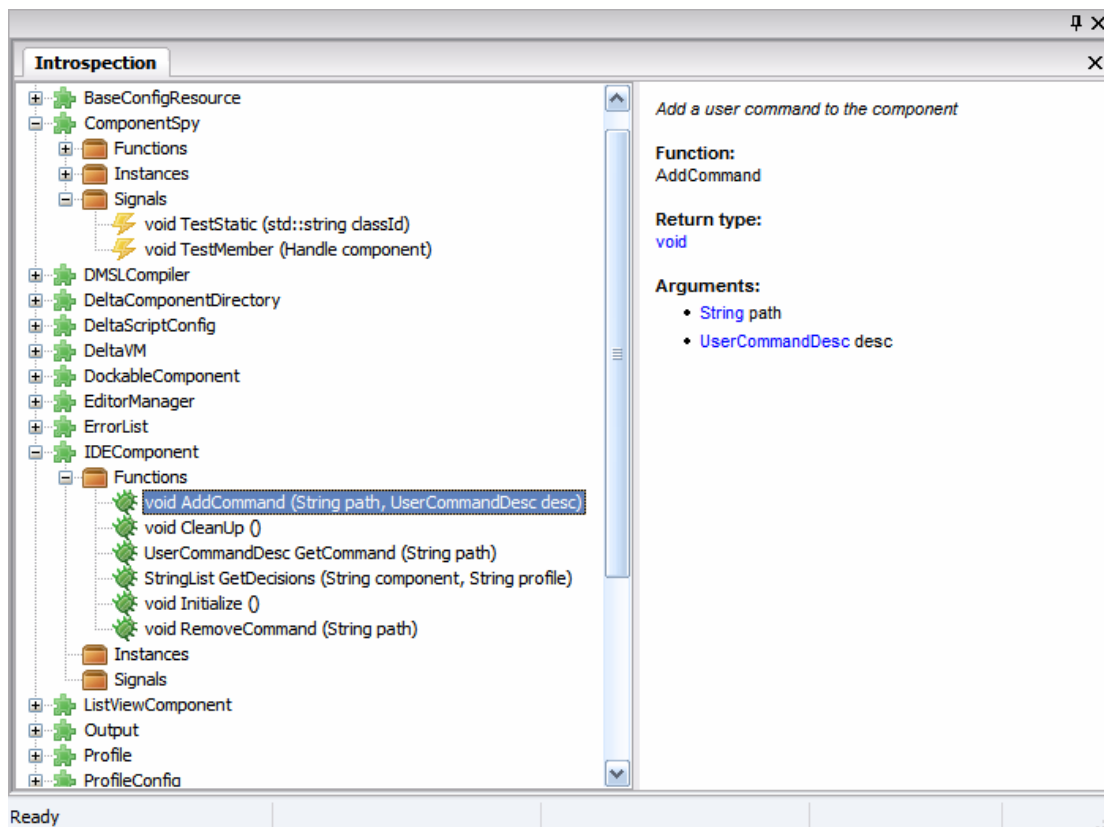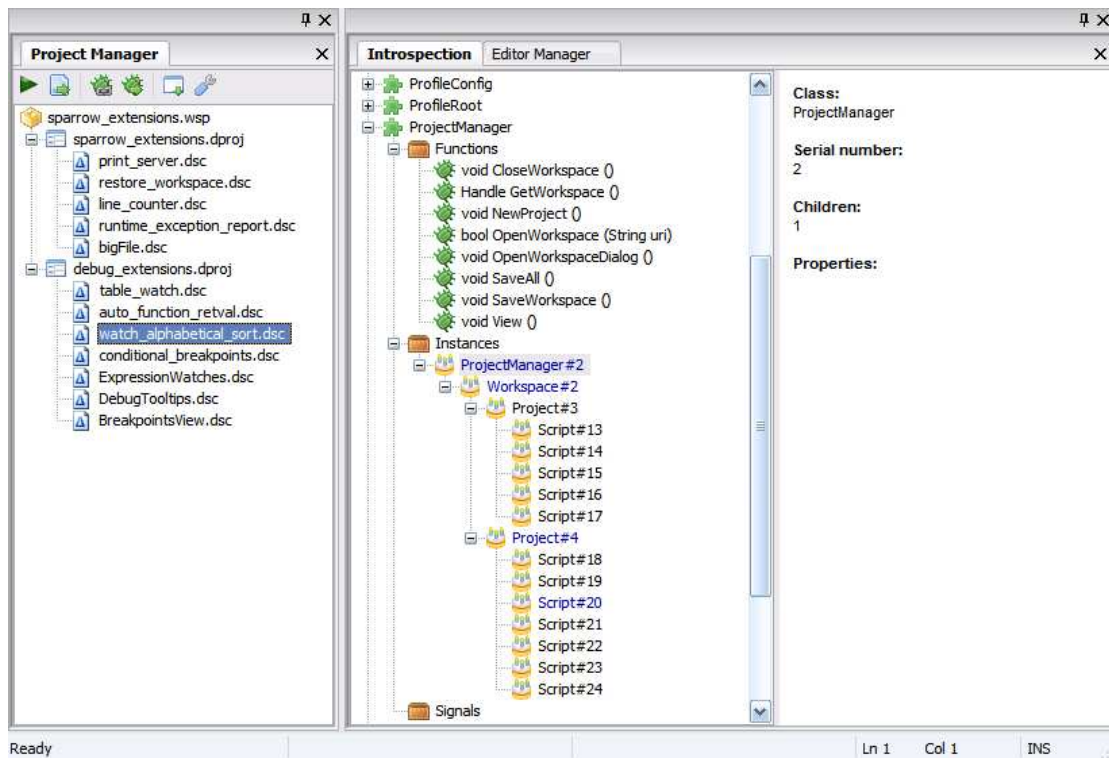**Figure 17** – Displaying the data of a component

Figure 18 – Displaying the documentation of a component function

Figure 19 - Displaying a component instance hierarchy

6. Syntax Directed Editor

The central part of any IDE is its source code editor. This is the tool that programmers use most of the time when developing applications. The role of an editor is to assist the programmer in the process of source code authoring. Typically, source code editor implementations in contemporary IDEs assist the programmer by providing the following facilities:

- Syntax Highlighting
- Code Folding
- Auto-Completion
- Syntax Validation

Sparrow's source code editor is, by design, language-agnostic, i.e., it is not restricted to editing only Delta code. On the other hand, the primary target of the IDE was Delta, and as such, Delta is the language with the most supported features. However, Sparrow's source code editor can be straightforwardly extended to provide more enhancements for other programming languages. Hence, the editor supports all the aforementioned facilities for the Delta language. For other languages it supports only syntax highlighting and code folding. Its main contributions, however, are the real-time syntax validation of the source code, the maintenance of the complete structure of the source code in an Abstract Syntax Tree form, and the ability to parse only the affected segments of the text during an edit operation – instead of parsing the whole file. The latter is the common approach that editors follow in order to support syntax validation. In the following sections, the architecture, implementation, and the most important features of Sparrow's editor will be presented.

6.1 Architecture

Sparrow's editor is implemented as a component and, thus, exports its interface to other components and to the applications that deploy the IDE. In order for the editor component to implement the functionality and user interface of a source code editor, it uses the "Editor Base" library that was implemented specifically for Sparrow but at

the same time is not bound to it. Therefore, the core functionality of the editor is provided by the "Editor Base" library. The editor component constitutes a lightweight wrapper for the editing interface that is exposed by the library and has the following responsibilities: (a) it exposes the configuration options as component properties (see section 3.3) and uses them to affect the functionality of the library, (b) it reads the language descriptions that are contained in a configuration file and configures the library accordingly, and (c) it emits signals to notify other components about the occurrence of significant editing events (e.g., the current line and column of the cursor, whether the file is modified, etc.)

The "Editor Base" library is based on and extends the Scintilla editing framework [14]. On the one hand, it wraps Scintilla in a class that simplifies the most common tasks and exposes a uniform configuration interface for affecting its functionality. On the other hand, the "Editor Base" library implements a plug-in mechanism – orthogonal to Scintilla's functionality – that enables the extensibility of every aspect of the editor's functionality through dynamically loaded plug-ins. Scintilla, by default, supports only the modular handling of syntax highlighting and code folding that are implemented by independent pieces of code, called "Lexers." The library, of course, maintains this mechanism for supporting the aforementioned functionality in languages for which Scintilla includes suitable Lexers, but, at the same time, through the plug-in interface, allows for many more language-specific adaptations. Sparrow's editor decides on which Lexer to install and/or which language module to load by looking at the extension of the edited file and executing the instructions that are included in its "Language Descriptions" file – an XML encoded configuration that describes the supported editor languages. All these architectural elements are displayed in Figure 20.
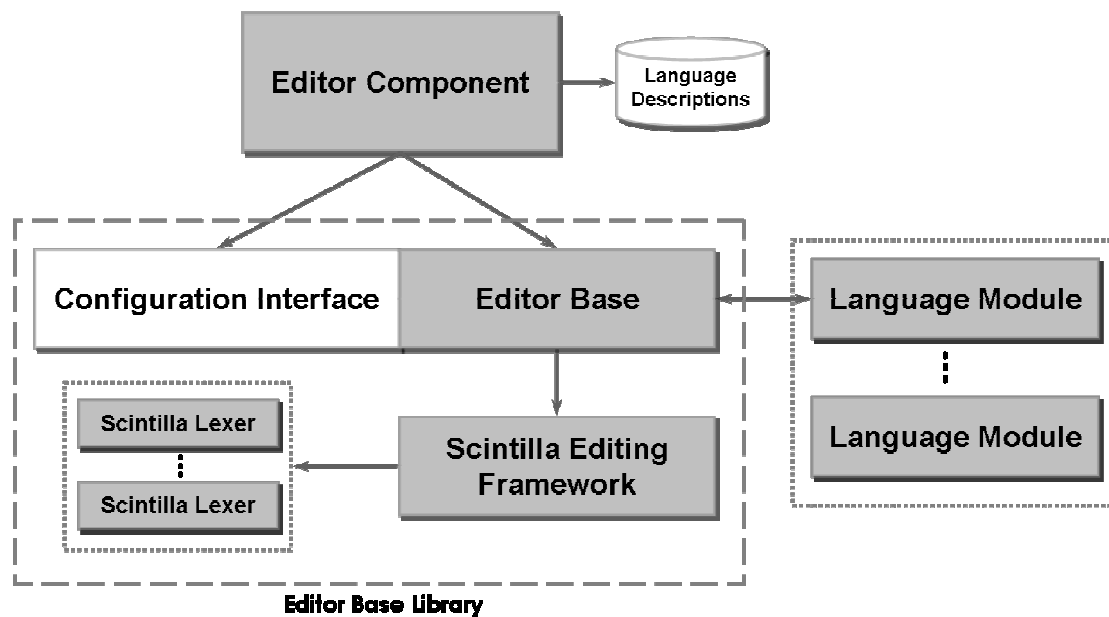
Figure 20 - Architecture of Sparrow's Source Editor

For the support of the Delta programming language, the plug-in functionality of the "Editor Base Library" is exclusively used. Thus, the rest of this chapter will focus on the design and the implementation of Delta's language module, which, in essence, implements all the editor features that were mentioned at the beginning of this chapter for the Delta programming language.

In Figure 21 the architecture of Delta's editor language module is displayed. The Delta Editor Interface is the entry point for the invocation of the module's functions from the "Editor Base" library. The Program Description element is the structure that holds a hierarchical representation of the Delta program (see section 6.4.) The Delta Parser is the unit responsible for parsing the editor's text and producing a convenient representation of the contained Delta program, i.e. a "Program Description" structure. Lastly, the Abstract Syntax Tree (see section 6.3) Visualizer and the Delta Scintilla Styler are responsible for providing a visualization of the program's structure and affecting the visual representation of the source code respectively. More details about the role of each of these elements will be presented throughout the subsequent sections.
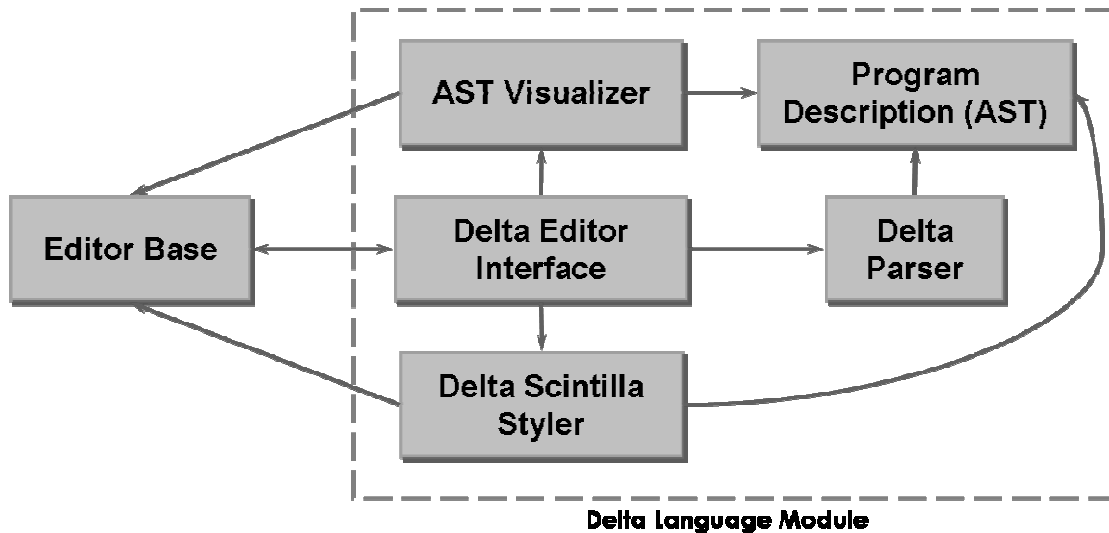
Figure 21 - Architecture of Editor's extension plug-in for Delta

6.2 Grammar Overview

As far as the editor is concerned, the most critical aspect of Delta is its grammar. The top level rules of Delta's grammar, as well as an example that conforms to each rule can be seen in Figure 22. The complete reference of Delta's grammar can be found in [23]. A delta program is essentially a set of "statements." The realization of this decomposition is essential for the implementation of the incremental parsing functionality of the syntax-directed editor (see section 6.4.)

```
Program:     ε
             | Stmts
             ;

Stmts:       Stmts Stmt
             | Stmt
             ;

Stmt:        Expression ';'      // e.g. a = 2 * c.pi * circle["radius"];
             | AssertStmt        // e.g. assert a and b or c;
             | WhileStmt         // e.g. while (false == true) {}
             | ForStmt           // e.g. for (i = 0; i < 1821; ++i) {}
             | IfStmt            // e.g. if (a / 5 == 3) {}
             | ReturnStmt        // e.g. return back;
             | Compound          // e.g. {}
             | LoopCtrlStmt ';'  // e.g. continue;
             | TryStmt           // e.g. try foo() trap exception {}
             | ThrowStmt         // e.g. throw this;
             | Function          // e.g. function foo (arg1, arg1) {}
             | ';'               // e.g. ;
             ;
```

Figure 22 - The top level rules of Delta's grammar in BNF

6.3 Abstract Syntax Trees

An Abstract Syntax Tree (AST) is a finite directed, acyclic, tree data structure, where each parent node denotes a language operator and each of its child nodes represent its operands. ASTs are very popular as a means of representing the hierarchical structure of a language's source. As such, they are ubiquitous as an intermediate representation of a program in compilers, where they are used for performing optimizations and producing the final, executable, code of the compiled program.

Whereas, typically, ASTs do not contain nodes that represent syntactic constructs that do not affect the semantics of the program, the implementation reported here contains them. This "lossless" representation of a Delta program was deemed important during the design of the editor, since future extensions may introduce refactoring or formatting tools for which the complete syntactic representation of a program is essential.

An exhaustive list of the AST nodes that are used for the representation of any Delta program can be seen in Table 4. Apparently, nodes that represent more than one syntactic constructs contain enough information so that they can be uniquely identified. Additionally, an example AST representation of a simple Delta program can be seen in Figure 23.

Table 4 - The AST nodes used in the representation of a Delta program

AST Node	Description
StmtsASTNode	A set of Delta statements
ExpressionListASTNode	An expression list
ArgListASTNode	An argument list (a list of ids)
UnaryKwdASTNode	All unary keywords (e.g. assert)
LeafKwdASTNode	All leaf keywords (e.g. break)
WhileASTNode	While statement
ForASTNode	For statement
IfASTNode	If statement
ElseASTNode	The "else" part of an if statement
CompoundASTNode	A list of statements enclosed in '{' '}'
TryASTNode	Try statement

`TrapASTNode`	The "trap" part of a try statement
`FunctionASTNode`	A function definition
`FunctionNameASTNode`	The name of a function's definition
`TernaryASTNode`	The ternary operator (a ? b : c)
`PrefixOpASTNode`	All prefix operators (e.g. prefix ++)
`SuffixOpASTNode`	All suffix operators (e.g. suffix --)
`BinaryOpASTNode`	All binary operators (e.g. +)
`UnaryOpASTNode`	All unary operators (e.g. unary -)
`CallASTNode`	A call expression
`VariableASTNode`	A variable instantiation
`ConstASTNode`	A constant expression
`ArgASTNode`	An argument (id)
`TableElemASTNode`	A table element
`TableElemsASTNode`	A list of table elements
`TableIndexListASTNode`	A table indexed list
`TableConstructASTNode`	A table construction expression
`TableConstKeyASTNode`	A constant table key
`OtherStmtASTNode`	A poorly named expression statement (e.g. a = 3;)
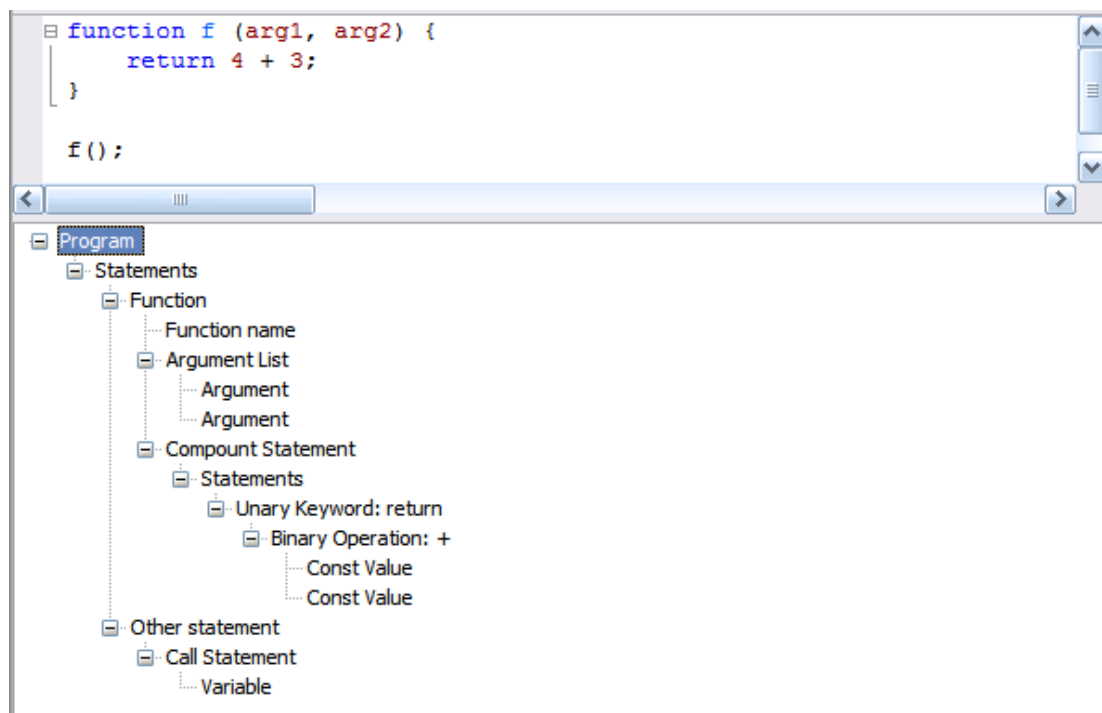
Figure 23 - Abstract syntax tree for a simple Delta program

One last issue worth noting is that the program's representation as a tree structure simplifies considerably its manipulation. Using the Visitor pattern [8] on the AST structure provides an intuitive way for interacting with the program's representation. Furthermore, the code that manipulates the AST is separated and modularized effectively without resorting to tedious and error-prone switch-case constructs.

53

6.4 Incremental Parsing

The Delta parser, along with the "Program Description" structure, is the central piece of the Delta language module. It parses a stream of text and produces a "Program Description" instance that contains a complete, easily processed, view of the Delta program that is enclosed in the editor. Specifically, the program description structure contains the following:

- The AST of the Delta program that was parsed successfully
- A list of parse errors that were encountered while parsing the Delta program
- A list of comments that appear in the program
- A list of text excerpts that could not be parsed

All these elements encapsulate a range structure that denotes the absolute positions of the referenced syntactical constructs inside the editor. The difference between the list of parse errors and the list of excerpts that could not be parsed is that the former contains only the text that triggered the error, whereas the latter includes also the text that was discarded by the parser in order to continue parsing from a consistent state; hence, the former is a subset of the latter.

The parser is implemented using the Bison parser generator [10] and the Flex lexical scanner generator [9]. Bison's grammar rules for the Delta language are responsible for constructing AST nodes and building the resulting AST, bottom-up, as the grammatical rules are being recognized. Flex generates the lexical scanner as a C++ class so that it can read its input stream from a standard C++ input stream (std::istream.) Sparrow's editor provides a specialization of the input stream for enabling clients to read the contents of the editor through the standard well-known interface of a C++ input stream. Thus, the resulting parser is able to parse the text directly from the editor's buffer.

The main trait of syntax directed editing is the maintenance of the edited program's structure in a format that can be easily manipulated programmatically. As such, Sparrow's editor permits the free-editing of the editor's text, while maintaining a consistent view of the program in a "Program Description" structure.

A trivial method for achieving the aforementioned functionality would be the reevaluation of the whole text of the editor after every single change. Apparently, "changes" include the insertion/deletion of characters by the user, the invocation of cut/paste commands, and the insertion/deletion of text programmatically (by other components or applications that deploy Sparrow.) This is a perfectly viable option when the edited files are kept small, i.e., less than 7,000 Lines of Code (LOC.) However, when files get bigger, the responsiveness of the editor deteriorates significantly; even more so, when the visualization of the program's AST is active (see section 6.5.4.) This decline in responsiveness is also evident in the editors of other IDEs that maintain internally a hierarchical representation of the program's structure, e.g., in Eclipse's Java source code editor. Additionally, it is important to note that an optimization in the speed of the – already fast – Bison/Flex generated parser would make no difference, as the primary bottleneck of the evaluation process is the generation of the graphical elements of the AST visualization.

Consequently, the editor's extension for the Delta language succeeds in eliminating the decrease in responsiveness when editing large files by evaluating after each change only the parts of the text that are affected by it. The resulting representation is identical to the representation that would be generated if the whole text was reevaluated; that is when the text constitutes a correct Delta program. When the parsed text contains an error, in which case the entire text is not a valid Delta program either, the resulting representation differs. An example of this "inconsistency" can be seen in Figure 24, where the user has just entered the text "f = /*" in the editor. However, these inconsistencies, in case of input that does not conform to the Delta grammar, do not render the incremental parsing approach inferior to the full-parsing one. In fact, the resulting representation in the case of partial-parsing is preferable, as it constrains potential errors in a smaller text area.
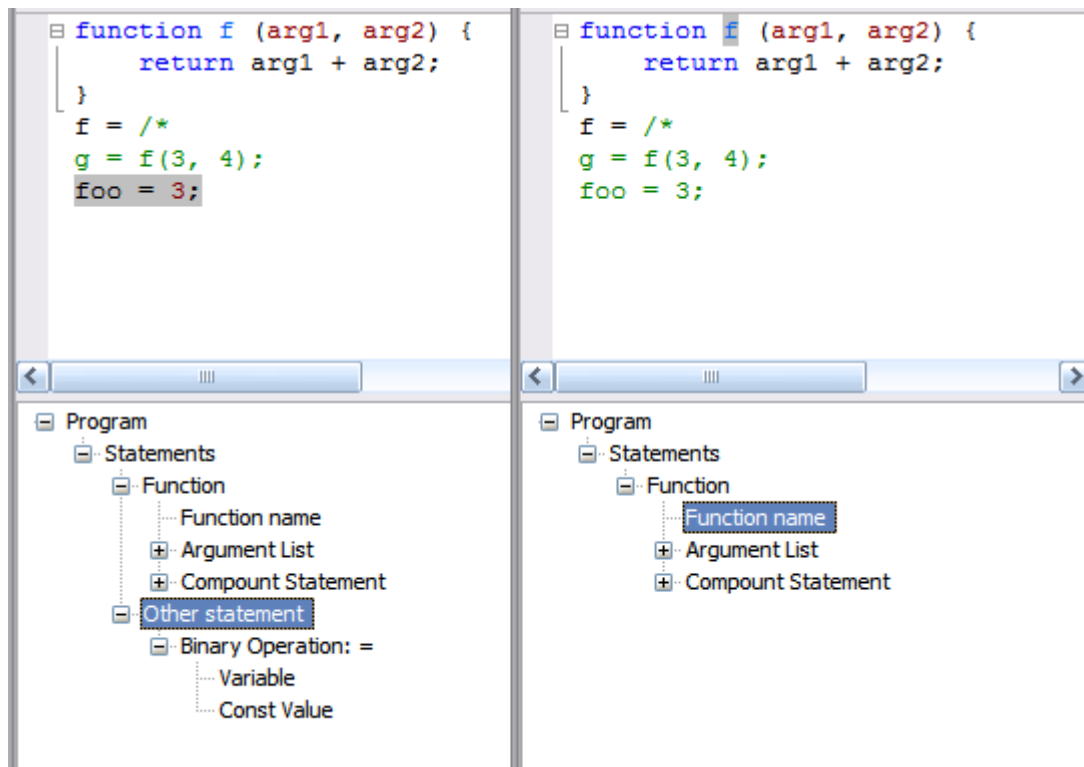
Figure 24 - Difference in AST in case of error; left: incremental parsing, right: full parsing

That said, the algorithm used by the Delta language support module to partially parse only the affected text after every modification, comprises of the following steps:

- Update the absolute position of all the elements that are affected by the change
  - Update the position of the AST nodes
  - Update the position of the errors
  - Update the position of the text that could not be parsed
  - Update the position of the comments
- Remove all the adjacent elements that are affected by the change
  - Remove the minimum Delta statement that contains the changed text
  - If there is text that could not be parsed *before* the statement
    - Remove it and also remove the statement *before* it
  - Otherwise, remove also the statement *before* the minimum statement
  - If there is text that could not be parsed *after* the statement
    - Remove it and also remove the statement *after* it
  - Otherwise, remove also the statement *after* the minimum statement
  - Remove all the comments and errors that are contained in the region of the removed elements

- Parse only the minimum region that contains the removed elements
- Merge the resulting representation of the newly parsed region with the main representation

An example execution of the algorithm can be seen in Figure 25, where the representation of the Delta program can be seen before and after the pasting of the "else" text. The stylish green underline marks the region of the text that is parsed after a text modification. The marking of the regions of text that are parsed after a modification is, by default, disabled; it can be enabled, however, by pressing simultaneously the keys Alt-Control-P.

Obviously, the operations that change the appearance of the edited text (i.e. syntax highlighting and folding,) as well as the update of the visual representation of the AST happen only in the incrementally parsed region.
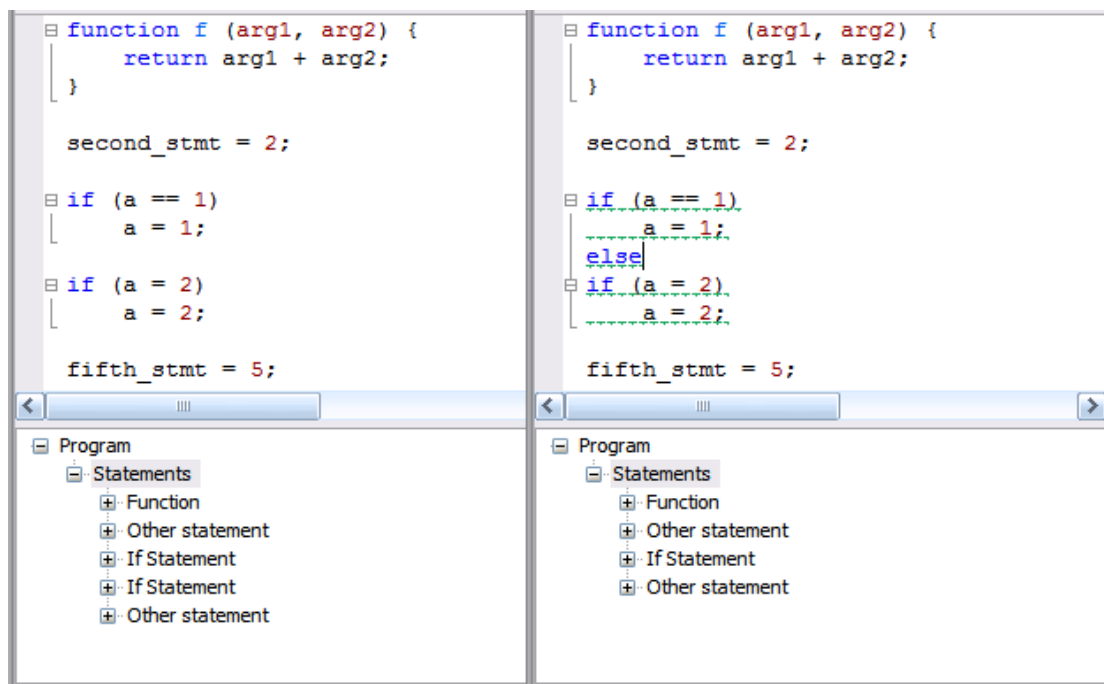
Figure 25 - Incremental parsing of text after pasting "else"

6.5 Rendering

6.5.1 Highlighting

Syntax highlighting is a function supported by all contemporary editors. It essentially varies the visual style of text excerpts that represent different syntactical structures of the target language. To be precise, syntax highlighting differentiates the visual representation of the types of the lexicographical tokens that constitute the "vocabulary" of a programming language.

Despite the fact that other editors, e.g., Visual Studio's and Eclipse's source code editors, base their highlighting on the recognized lexicographical tokens, Sparrow editor's Delta extension highlights the different language constructs based on their syntactical context. For Delta, hence, identical lexicographical tokens can have a completely different style depending on what they represent in a Delta program. E.g., in Figure 26, ids that appear as "object members" have different color from the ids that appear as variables. Additionally, in the same example, strings that appear as "table keys" (which is actually another way of accessing "object members,") have different color from strings that appear as plain expressions.

Syntax highlighting is implemented by an AST traversal that applies the corresponding style on the text whose location is indicated by the absolute positions that are encapsulated in AST nodes. Comments, that are separate from the AST, are styled by traversing the comment list of the "Program Description" structure and the parts of text that could not be parsed are styled lexicographically. This lexicographical styling is essential for giving immediate feedback to programmers when they are typing; otherwise, e.g., the "if" keyword of an incomplete "if" construct would be highlighted only after it became a complete statement and not as soon as it was typed.

6.5.2 Error Marking

Error marking refers to the ability of the editor to recognize and mark the syntactical errors that are present in the source code of a program, i.e., the parts of the program that do not conform to the target language's grammar. Syntax validation is usually implemented, in editors that support it, with the help of the language's compiler.

Visual Studio for the .NET languages and Eclipse for the Java language support syntax validation by invoking the compiler with the edited file as input, and obtain from it the parts of the text that do not conform to the grammar. Although this method can also recognize semantic errors, it is not immediate. The validation of the program is deferred until its compilation. In Sparrow's Delta editor, syntactic errors are marked as such instantly – as the user types.

As seen in Figure 26, errors are indicated by a wavy red underline. The implementation of the error marking functionality is straightforward: the contents of the error list field of the "Program Description" structure (see section 6.4) are styled with the red underline.
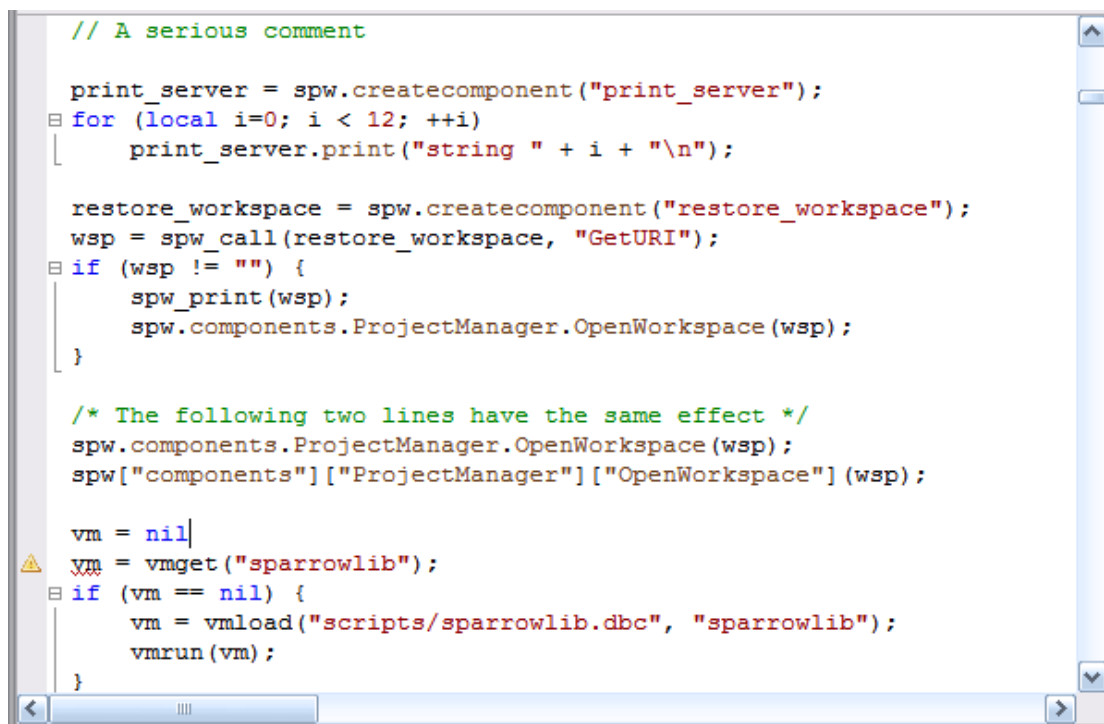

Figure 26 - Editor Syntax highlighting

6.5.3 Code Unit Folding

Code unit folding provides the means for programmers to toggle the visibility of source code segments corresponding to specific syntactic elements. Its typical usage is

hiding the body of functions or classes so that programmers can concentrate undistracted on the parts of the source code they are editing.

Sparrow's Delta editor supports code folding for the following constructs:
- While statements
- For statements
- If – Else statements
- Try statements
- Function definitions
- Table construction statements

Code folding is again implemented by an AST traversal, so the syntactic context of the structures that can be "folded" is known, and thus, it does not depend on isolated character sequences.

6.5.4 AST View

This facility of the Delta editor is visible in many of this chapter's figures (e.g. see Figure 23.) Users can toggle the visibility of the AST visualization window by pressing the keys: Alt-Control-V in the editor when editing a delta source file.

In addition to providing visualization of the internal AST of the edited program, this tree is also interactive. Selecting any visualized node selects its corresponding text in the editor. Right-clicking on a node, allows the user to perform any of the following actions: (a) delete the node's corresponding text, (b) copy the node's corresponding text, or (c) cut the node's corresponding text.
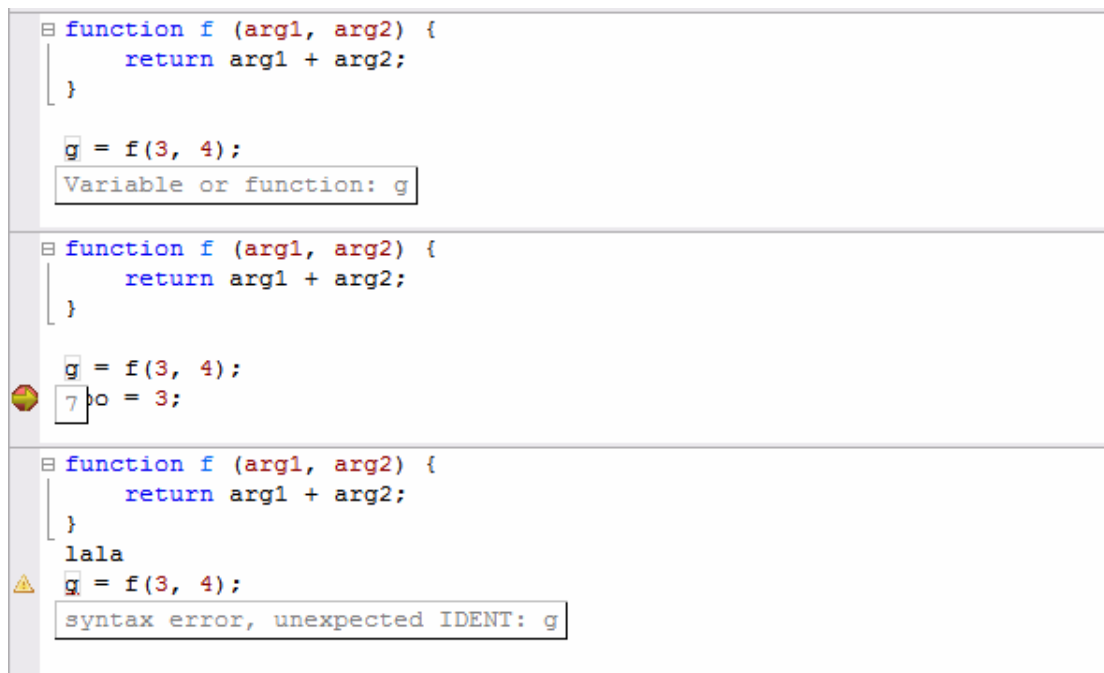
6.5.5 Tooltips

Another feature of the editor's extension for the Delta language is the provision of context and function sensitive tooltips when the mouse pointer remains over the editor's text for a couple of seconds. When that happens, the AST node that

corresponds to the text under the mouse pointer is retrieved and information for that node appears in the form of a tooltip. The text displayed by the tooltip can be modified by extensions.

So, in the default case – where the user is editing text – the displayed tooltips give information on the role of that text in the Delta program, i.e., they display the type of the text's AST node. When the mouse pointer is dwelled over text that is marked as a syntax error, the explanation of that error – that is obtained from the generated Bison parser – is displayed. Lastly, one of the extensions of the Delta debugger [5] takes advantage of this functionality to display the value of variables during debugging. Figure 27 illustrates the different realizations of the editor's tooltips.

```
□ function f (arg1, arg2) {
      return arg1 + arg2;
  }

  g = f(3, 4);
  Variable or function: g
```
```
□ function f (arg1, arg2) {
      return arg1 + arg2;
  }

  g = f(3, 4);
  7 bo = 3;
```
```
□ function f (arg1, arg2) {
      return arg1 + arg2;
  }
  lala
  g = f(3, 4);
  syntax error, unexpected IDENT: g
```

Figure 27 - Editor Tooltips under the mouse pointer

6.6 Auto Completion

Another universally supported function is the Auto-completion of symbols, which refers to the automatic suggestion of valid symbols that may appear at a specific position in the text of a program. Usually, editors of contemporary IDEs present the suggested symbols in a drop-down selection list.

Automatic completion of all the valid symbols in dynamic languages such as Delta is a very difficult issue. Delta's objects can be extended at runtime by adding member variables and functions. This situation is further complicated by the fact that even the inheritance of Delta objects is a runtime function [23]. This essentially means that is practically not possible to know an object's members without actually running the complete Delta program.

Instead of resorting to semi-accurate heuristics to infer the member data of a Delta object, it was decided to offer a completion list of the symbols that were previously used *above* the point where the auto-completion takes place.

The Delta editor distinguishes four kinds of symbols:
- Object members (or object keys)
- Functions
- Function arguments
- Variables

So, when the user presses the dot key ('.') right of an id, all the previously accessed object members are retrieved (by traversing the AST "upwards") and presented in a drop-down list. As mentioned previously, the notation "object.member" in Delta, is just a shorthand for the expression 'object["member"];' thus when the user types the text '["', all the object members are, similarly, retrieved but this time they are offered as strings instead of plain ids. Additionally, when the user starts typing a word whose first three letters match existing symbols of functions, function arguments, or variables, their names are offered for auto-completion. The completion list for the aforementioned cases, which can also be activated by pressing the keys Control-Space, can be seen in Figure 28.

Figure 28 - Automatic completion of symbols, and object members as ids and strings

7. Summary and Conclusions

7.1 Summary

In this Thesis, we have presented a large part of Sparrow, a circular meta-IDE for the dynamic object-based language, Delta. Initially, we have discussed the overall component architecture and the basic facilities built on top of it, like the undo manager and the introspector. In this context, we have outlined the primary design philosophy regarding architectural openness, dynamic extensibility, and customized programmable deployment. Additionally, we have shown particular component features like the support for self-embedded documentation and the introspection interface. Finally, we have presented the source-level editor with its multi-language open architecture, putting emphasis on the implementation of the plug-in to accommodate the Delta language.

7.2 Conclusions

Our choice to implement the Sparrow IDE has been dictated from various technical reasons, linking directly to the main objectives of providing a dynamic, circular and extensible platform. More specifically, the need to offer dynamic query and introspection of components during runtime implied that a comprehensive component technology could be deployed with these qualities, and we had a few choices for that, with the most easy to use alternative being Java and OSGi. However, since we wanted to support dynamic extensions of component methods as a standard feature for components, a method we called vertical extensibility, we could not use such technologies as they require that such facilities are manually introduced by component vendors. Finally, we wanted to allow dynamic inheritance among components, i.e. runtime inheritance, something that is not offered by any componentware technology. Overall, while our emphasis has been to offer a typical component infrastructure, our focus was shifted towards specific features relating to openness and extensibility. In this sense, our component system does not aim to compete with existing technologies, but to offer a layer of functionality optimally fit

to the needs of a dynamically extensible and deployable IDE. The latter implied that our work could not be hosted by popular IDEs that genuinely rely on such technologies, as it is the case with Visual Studio (COM backbone) or Eclipse (OSGi backbone.) Now, the effort to introduce on top of such technologies facilities like component editing and inheritance was far beyond our objectives. Hence, we preferred to focus on a new compact component subsystem where the specific required functionality would be put at place by design, rather than introduced as an afterthought.

Along these lines, the remote deployment API, as well as the interactive introspection had to be built on top of the basic component infrastructure. Consequently, we introduced these facilities as extensions of our component subsystem: (a) introduce inter-process dispatching of method invocations for remotely deployed components, and (b) provide a direct manipulation interface to the APIs of the active components. One of the future extensions regarding the component system is to introduce *dynamic inter-process inheritance* among components, meaning a local component may inherit on-the-fly from a remote one. In our current implementation, the latter scenario is supported only when both components reside at the same process space, i.e. *dynamic intra-process inheritance*.

One module that could be potentially implemented on top of an existing IDE platform is the source code editor. Nevertheless, such an approach does not essentially reduce the required code size, since it still requires an implementation from scratch of the most critical editor part: the syntax-aware editing functionality for the Delta language. Practically, to automate the latter it would imply the presence of *true syntax-highlighter editor generators*, analogous to parser generators, the former tools currently missing. We put particular emphasis on *true syntax highlighting*, since although all existing editors offering highlighting configuration merely support lexical highlighting, the latter is improperly referred to as syntax highlighting. So, in any case, the implementation of this plug-in would be, for the most part, invariable. The graphical rendering of the editor and its editing functions that would have been provided by an underlying platform are supplied, in our implementation, by the Scintilla framework. So, in this case, we wouldn't have gained anything by basing the editor on the facilities provided by other IDEs.

In addition to that, Sparrow will be used as an internal development tool for the software development division of the Human Computer Interaction (HCI) laboratory of the Institute of Computer Science (ICS) at the Foundation for Research and Technology – Hellas (FORTH.) Thus, the decision not to base Sparrow on top of an existing platform was further supported by the strategic decisions of the laboratory regarding the usage of the Delta programming language. The independence of the laboratory from external software systems was deemed essential for the deployment of Delta in future projects.

In closing, we shall mention that our architectural and implementation strategies for Sparrow, as a component-oriented platform that can easily accommodate diverse, domain-specific, functionality, proved to be successful. They proved to be a robust approach to the development of a large scale, extensible and adaptable system such as Sparrow; and the proof for this is the resulting, fully functional, Integrated Development Environment for the Delta language.

References

[1] David Abrahams and Aleksey Gurtovoy: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley, Reading, MA, 2004. ISBN 0-321-22725-5

[2] Andrei Alexandrescu: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, Reading, MA, 2001. ISBN 0-201-70431-5

[3] David M. Beazley, SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++, 4th Annual Tcl/Tk Workshop, Monterey, CA., 1996

[4] Boost C++ Libraries. http://www.boost.org

[5] Themistoklis Bourdenas: Circular Meta-IDE for the Delta Language: Extensibility Layer for Delta, Debugger, Runtime adaptation, and Project Manager. Master's Thesis, 2007

[6] D. Box: Essential COM, Addison-Wesley, 1998. ISBN 0-201-63446-5

[7] Brad J. Cox, Andrew J. Novobilski: Object-Oriented Programming: An Evolutionary Approach. 2nd ed. Addison-Wesley, Reading, MA, 1991. ISBN 0-201-54834-8

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995

[9] Flex: The Fast Lexical Analyzer. http://flex.sourceforge.net

[10] Free Software Foundation: Bison – GNU Parser Generator. http://www.gnu.org/software/bison

[11] Grimes, Richard: "ATL and COM", ATL COM, 1st edition, Wrox Press, 1998. ISBN 1-861002-4-91

[12] Microsoft Corporation: Dynamic-Link Libraries. http://msdn2.microsoft.com/en-us/library/ms682589.aspx

[13] Microsoft Corporation: Visual Studio Integrated Development Environment. http://msdn2.microsoft.com/en-us/vstudio/default.aspx

[14] Neil Hodgson: Scintilla source code editing component. http://www.scintilla.org

[15] K Desktop Environment (KDE): KDE API Reference: The DCOP Desktop Communication Protocol Library. http://api.kde.org/3.5-api/kdelibs-apidocs/dcop/html/index.html

[16] K Desktop Environment (KDE): KDevelop. http://www.kdevelop.org

[17] K Desktop Environment (KDE): KParts: Creating and Using Components. http://developer.kde.org/documentation/tutorials/kparts

[18] Bertrand Meyer: Object-Oriented Software Construction. 2nd ed. Prentice Hall, 1997. ISBN 0-136-29155-4

[19] Object Management Group: The Common Request Broker Architecture Specification. http://www.omg.org/technology/documents/formal

[20] William F. Opdyke: Refactoring Object-Oriented Frameworks. PhD Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1997

[21] OSGi Alliance: OSGi Service Platform – Release 4. (2005)

[22] Python Software Foundation: Python Programming Language. http://www.python.org/

[23] Anthony Savidis: Dynamic Imperative Languages for Runtime Extensible Semantics and Polymorphic Meta-Programming. RISE 2005: 113-128

[24] Sun Microsystems: Java Remote Method Invocation (Java RMI) Specification. http://java.sun.com/j2se/1.5.0/docs/guide/rmi/index.html

[25] Sun Microsystems: JavaBeans 1.01 specification. http://java.sun.com/products/javabeans/docs/spec.html

[26] The Eclipse Foundation: Eclipse Project. http://www.eclipse.org

[27] The Mozilla Foundation: XPCOM (Cross Platform Component Object Model) Reference. http://www.xulplanet.com/references/xpcomref

[28] Trolltech: Qt: Cross-Platform Rich Client Development Framework. http://trolltech.com/products/qt/

[29] wxWidgets GUI toolkit, http://www.wxwidgets.org