An Adaptable Circular Meta-IDE for a Dynamic Programming Language

Anthony Savidis\textsuperscript{1,2}, Themistoklis Bourdenas\textsuperscript{1}, Yannis Georgalis\textsuperscript{1}

\textsuperscript{1} Institute of Computer Science, Foundation for Research and Technology – Hellas,
\textsuperscript{2} Department of Computer Science, University of Crete
\{as, themis, jgeorgal\}@ics.forth.gr

Abstract. Integrated Development Environments (IDEs) are amongst the most popular programming tools, significantly judging the usability, acceptance and popularity of programming languages. Recent trends, reflected in widely used IDEs such as Visual Studio and Eclipse, besides the typical multi-language support, emphasize openness through extensibility and deployment by exposing intrinsic APIs. In this context, we discuss the development of the Sparrow IDE for the Delta dynamic object-based language, supporting some new features that we consider beneficial to programmers: (a) dynamic adaptations and configurations through rules and profiles; (b) extensibility both vertically - replacing or modifying functionality, and horizontally - introducing functionality; (c) circular development - write programs using the IDE which extend and manage the IDE itself; and (d) customized extensibility and deployment to build application-oriented development environments, thus treating Sparrow as a meta-IDE.

Introduction

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 comes to medium-to large scale systems. Most recently, IDEs like Visual Studio (Microsoft, 2005) and Eclipse (Winmad, 2003), are made mostly as open collections of task-oriented tools, gathered and managed together under an extensible programming environment, rather than monolithic systems of the all-in-one style. In this context, sparrow is an IDE for the Delta dynamic object-based language, developed following three key objectives:

- To allow programmers express and apply ways of adapting the IDE according to their individual programming habits, i.e. it is an adaptable IDE.
- To support extensibility of features, allowing such extensions to be developed directly using Sparrow itself, i.e. it is a circular IDE.
- To facilitate open deployment by third-party application development environments so as to build domain-oriented IDEs, i.e. it is a meta-IDE.
An important aspect of the Sparrow development process has been the extensive application of the circular development style. More specifically, for every system component an initial basic version was originally implemented in C++. Subsequently, once this component was tested and incorporated to the Sparrow IDE, the rest of its functionality was implemented through the Delta language and using the Sparrow IDE itself. Effectively, the Sparrow IDE makes available to its users all of its current, as well as future components, as a runtime library available in the Delta language.

We continue by firstly introducing and briefly explaining the basic support to facilitate runtime adaptation, development circularity and domain-specific meta-extensions.

Supporting adaptations

Our particular notion of adaptation concerns the capability of the software system to carry out dynamic selection and activation of its components best fitting individual user profiles (Savidis et al, 2005). The latter, from the system point of view, requires a flexible and open architecture to practically enable dynamic system assembly from a pool of chosen software components (Savidis, 2004). We support two ways to express dynamic adaptation-oriented configurations that can be applied on-the-fly:

- Configuration management scripts implemented in the Delta language (Savidis, 2004).

Supporting IDE circularity

In the Sparrow IDE, circularity relates to extensibility support, enabling to introduce or update functionality dynamically as follows:

- As normal dynamically-linked libraries (DLLs), implemented in C++ on top of the Sparrow extension libraries (C++ version). We call this the native extension layer.
- As compiled Delta Virtual machine byte code, enabling extension components to be implemented in the Delta language using the Sparrow IDE and the Sparrow extension libraries (Delta version). We call this the circular extension layer.

Practically, the Sparrow IDE offers a very basic component skeleton, implementing essentially: (a) a User Interface container; and (b) a dynamic component infrastructure relying on component directories, message routing and component protocols. In the design of the system we adopted a tabula rasa concept, i.e. can make a system out of its skeleton, while even the basic components were built as extensions over the initial skeleton.
Supporting a meta-IDE

Naturally, one of the primary objectives for the development of the Sparrow IDE was to offer integrated support for the development of programs written in the Delta language, a dynamic object-based language (Savidis, 2004) for manipulating libraries offered by applications (host-systems following the scripting languages' terminology) implemented in C++. A system relying on Delta is Game Maker 1.0 (see Figure 2), being a domain-specific application development environment for cartoon-like games.

Figure 1: Supporting circular extensions by enabling new features of the IDE to be implemented using the IDE itself.

Figure 2: GameMaker IDE for games relying on the Delta language, encompassing all-in-one the authoring, programming and debugging facilities.
The basic game logic is programmed in the Delta language, while all facilities of the underlying C++ game engine are offered as Delta library functions. During game development, it has been constantly required to produce customized development facilities, like scene-based organization of game scripts, special debugging tool-tips for displaying the properties of sprite variables, customized compilation sequences, etc. In all such cases we were obliged to directly update the source code of the original system, something that was impractical for a number of reasons: (a) some updates were customized to individual programmers; (ii) the programmers of some modules were no longer working with us; and (iii) we did not like the idea of a monolithic tool constantly growing in size. Additionally, we had to implement all Delta development facilities as an integral part of the game development system. Once we very soon decided to deploy Delta in another domain (mobile location-aware information systems), we realised that replicating or trying to extrapolate such Delta development features was an imperfect choice.

In this context, we decided to develop the Sparrow IDE putting emphasis on facilitating the open deployment by third party tools so as to build domain-customized development environments. The idea is that for application systems relying on the Delta language (e.g. games, mobile applications, etc.), the Sparrow IDE should deliver the basic programming facilities, while enabling the incorporation of additional functionality through the development of extension components. This technical notion is illustrated under Figure 3.

For this purpose, the Sparrow IDE offers a generic deployment API, through which all third-party development environments have full programming access to all of its components. Additionally, every integrated component may be dynamically replaced, as far as it obeys the original API (runtime consistency), and it provides related semantic behavior (usage consistency). Two types of domain-specific extensions are supported in the Sparrow IDE:

- **Horizontal extensions**, i.e. introducing new types of functionality (e.g. call graphs visualizers, documentation generators, etc.)
- **Vertical extensions**, i.e. replacing components by alternative versions of the same functionality, or extending the original component functionality.

![Figure 3: Making domain-specific environments, for building applications relying on the Delta language, on top of the Sparrow IDE.](image)
This principle is outlined within Figure 4. As illustrated, the basic IDE comprises an open collection of components, enabling dynamic replacements (i.e. may switch a component on-the-fly, activating an alternative version) and modifications or customizations (i.e. configuring or extending a component by introducing additional related functionality).

Figure 4: Domain-specific IDE deployment and growth through horizontal and vertical component extensions.

Such types of extensions populate a component with extra related functionality, a feature commonly known as vertical growth or extension. Additionally, reflecting particular application-oriented authoring or development requirements, new layers of functionality may be incorporated by programmers as additional components, something that is characterized as system horizontal growth or extension.

Related work

The notion of circularity is frequently met in programming and software engineering with varying connotations. Probably the most typical case concerns bootstrapping, related to the problem of producing systems that should eventually rely on their own functionality (like operating systems, language compilers, or virtual machines). We focus on circularity of IDEs to imply that they can be extended by programs built using the IDE itself. Such a feature is supported through plug-in mechanisms by modern IDEs like Visual Studio and Eclipse. However, it concerns only vertical system extensions, since only new components may be introduced. The manipulation or extension of IDE constituent components is not supported.

It important to note that the Sparrow IDE offers access to all available components through a component meta-API supporting introspection as well. In particular, once new components become an integral part of the Sparrow IDE, they are directly exposed to programmers through the meta-API. Regarding the objective of producing a meta-IDE, we were initially targeted in the provision of facilities enabling domain-
specific incarnations of the Sparrow IDE, however, only as a development environment for the Delta language. Initially, it was not in our primary design intentions to make the Sparrow IDE play the role of a generic IDE for producing all sorts of customized IDEs out of it, neither to encompass multi-language support. However, the way language-specific facilities have been incorporated allowed component extensions or replacements to introduce language-dependent functionality. For instance, the source-level debugger and editor are components entirely separating the User Interface logic from their underlying semantics. In particular, the editor relies on a tokenizer and parser producing an AST for syntax highlighting, while the debugger provides a graphical programmable place holder for watches, breakpoints, call stack management, and trace control, being entirely separated from the actual component implementing the debug logic and the language-specific version of the debugger API.

Besides circularity and meta-functionality, we should also mention the Klein programming environment (Ungar et al., 2006), a Self variant (Smith & Ungar, 1995), built as a meta-circular Virtual Machine (VM), i.e. the Klein VM is actually implemented in its own assembly language. In this context, the ‘meta’ property concerns circularity, as the Klein VM is not merely built in the Self language, which would imply circularity, but directly in its assembly language, meaning meta-circularity.

**System Architecture**

We briefly discuss the system architecture, discussing part of the software design regarding the component-based infrastructure of the Sparrow IDE (see Figure 5). More specifically, component class information regarding APIs, data properties, and construction / destruction methods resides within the **Component Class Registry, Metadata and Factory**. The latter is a singleton class, being the only global entry point to perform component instantiation and destruction. Component classes may be loaded dynamically by the **Component Class Loader**, either as DLLs or as Delta byte code, holding loading information for each dynamically loaded component class.

![Figure 5: The component-based infrastructure of the Sparrow IDE.](image-url)
In our architecture, component-class clients are implementationally decoupled from respective component classes, since component instance manipulation is possible only through the meta-API offered by the super-class Component Class. Additionally, the registry allows programmers to dynamically query components and extract their properties and methods, i.e. supports introspection. Moreover, this introspection facility is interactively provided to programmers, offering method invocation and documentation display (see Figure 6): (a) in case of methods with a list of arguments the programmer is prompted to provide values to all parameters (overloading is also supporting, enabling to choose among the alternative signatures); and (b) if there is a return value after invocation, it is displayed.

Figure 6: Interactive support for component instance introspection.

The macro-level architecture of the Sparrow IDE is provided under Figure 7. The basic User Interface is offered as an extensible graphical template supplying a main menu, toolbars, tab boxes, and tiled or docking windows. This template sets specific rules through which components introduce toolbar / menu items and tab areas in already existing tab lists. Additionally, multi-language extensions of the project manager, editor and debugger are effectively supported by the Sparrow IDE, as in most known IDEs, through appropriate interfaces (see API entries in Figure 7).

A snapshot of the basic Sparrow IDE is provided in Figure 8, showing the project manager, source editor with syntax highlighting, debugger while running a debug session, adaptations, and the directory of extension components implemented in the Delta language. The middle-right tab window provides a list of all IDE extension components (entitled Delta Component Directory) that are implemented using the
IDE itself (i.e. circular extensions). The user may select a component and activate it (instantiate) or close it (destroy). Interestingly, this particular component directory has been also implemented as an extension component.

![Diagram of Sparrow IDE components](image)

**Figure 7: Macro-level architecture showing multi-language extensibility.**

The snapshot of Figure 8 shows a debug session in progress, presenting active breakpoints and the call-stack. Additionally, the entries in the directory of Delta components marked with a green ‘tick in a circle’ on their left side indicate components that

![Screenshot of Sparrow IDE debug session](image)

**Figure 8: Overview of the Sparrow IDE showing various key components.**

The snapshot of Figure 8 shows a debug session in progress, presenting active breakpoints and the call-stack. Additionally, the entries in the directory of Delta components marked with a green ‘tick in a circle’ on their left side indicate components that
are currently activated (e.g. line_counter - line counter, and ExpressionWatches - expression watches), while the rest marked with a red ‘period in a circle’ concern components not yet active (e.g. auto_function_retval - the automatic function return value displayer, and runtime_exception_report - exception handling reporter).

Functionality for Adaptations

The support of adaptations enables programmers create rationalized configurations of the Sparrow IDE, effectively bridging user profiles with corresponding components. The adaptation mechanism relies on the DMSL language (Savidis et al., 2005) supporting the specification of user profiles, stereotypes (partial or entire valuations of the profile parameters) and selection rules per component class (see Figure 9).

```plaintext
stereotype DeltaBeginnerUser : params.user.delta_exp = beginner
stereotype DeltaAdvancedUser : params.user.delta_exp = advanced
stereotype Debug : params.user.debug = true
stereotype ExtensionAuthor : params.user.extension_author = true

component Sparrow { activate ProjectManager
activate Output if hasattr(user.delta_exp) and DeltaBeginnerUser then activate WelcomePage if hasattr(user.delta_exp) and DeltaAdvancedUser then {
   activate AdaptationManager
   activate ErrorList
}
if hasattr(user.debug) and Debug then {
   activate ExpressionWatches
   activate CallStackView
   activate BreakpointsView
}
if hasattr(user.extension_author) and ExtensionAuthor then {
   activate DeltaComponentDirectory
   activate ComponentSpy
}
}
```

Figure 9: Adaptation rules for component activation (top side), and the User Interface to handle profiles, adaptation rules and scripts (bottom side).

The adaptation rules encompass decisions that may concern different components of the Sparrow IDE. The evaluation starts by the top-level component, i.e. the Sparrow application object, which requires the evaluation of its block (see Figure 9, top part).
In this case, the evaluation will result in various activation decisions, which are appropriately dispatched using the Component Registry to instantiate objects of the corresponding component classes. Detailed rules may further be defined for distinct components; in this case, each created component is responsible to inquire the evaluation of its respective rule block using the DMSL interpreter.

Besides adaptation rules, programmers may define special-purpose scripts, handling components to bring dynamically the Sparrow IDE to a desirable required state, e.g. for debugging, for refactoring, for project creation, etc. Such scripts, being populated inside the Adaptations window (see Figure 9, bottom part), can be executed by the programmer on-demand.

**Functionality for Circular IDE**

As previously discussed, circularity means supporting Delta programs that can extend the Sparrow IDE itself. Such extensions should normally interoperate with other components like the source editor, the project manager and the main User Interface template. Additionally, extension components will have to deploy other components using the basic component API (defined in the component super-class), without requiring awareness of the component implementation form, e.g. whether it is in C++ or Delta. To allow components implemented in the Delta language be deployed in a way transparent to their implementation, we had do support cross-language method invocations (see Figure 10), i.e. calling from the Delta language the methods of other deployed components implemented in C++ or Delta, and vice versa.

More specifically, following Figure 10, the dispatch adapter maps a method invocation request originated from the Delta language to its corresponding C++ call using the component API (deploys bridge), while the invocation adapter maps a method invocation posted from C++ code to its corresponding Delta language call through the VM embedding API (deployed bridge). The latter embedding mechanism is similar to the one offered by other scripting languages like Lua (Ierusalimschy, 2003) or Python (Python, 2005).

![Figure 10: Method dispatching to support cross-language method invocations.](image-url)

Components implemented in the Delta language are effectively derived from the Component class by supporting reuse (calling base methods) through delegation and extension (introducing methods in Delta) through the invocation adapter (see Figure 11). In particular, every Delta component inherits Component class functionality by encompassing a delegate comp instance of the Component class.
function component_dispatcher(handle, index) {
  Initially check if index is an existing method
  local property = spw_getproperty(handle, index);
  if (property)
    return property;
  Treat the index as a function, binding spw_call()'s first arguments
  return bind(spw_call, handle, index);
}

components = [{ ".." : {
  function(t, classId) {
    Overload basic member access operator
    Get internal component handle from class id
    local handle = spw_getcomponent(classId);
    Install the method dispatcher to the component
    handle[".."] = component_dispatcher;
    return handle;
  }
}}];

A sample calling style, showing the typical OO syntactic style
components.ProjectManager.OpenWorkspace("sparrowlib.wsp");

Figure 12: Method dispatch adapter enabling invocation of base Component methods from within Delta code; the last line shows a typical call style.

Following Figure 12, the code enabling dispatching to Component base class methods while calling through an instance of a component implemented in the Delta language, relies on the support of the Delta language for dynamic overloading of the member access operator (double dot). More specifically, in place of the original member lookup function (dispatcher or virtual table), a new dispatcher is installed named component_dispatcher, the latter calling spw_getproperty library function of the Sparrow IDE to extract a property method from the component’s property dictionary.

Functionality for Meta IDE

Domain-oriented customization and growth of the Sparrow IDE is facilitated through component deployment, extension and adaptation. The Delta language version of the basic deployment API is provided under Figure 13. Functions spw_createcomponent and spw_destroycomponent, as expected, support the creation (from named classes) and destruction of component instances respectively during runtime.

handle spw_createcomponent (class_id, {handle | class_id})
void spw_destroycomponent ((handle | class_id))
void spw_setparent (handle, {handle | class_id})
handle spw_getcomponent (class_id[, serial])
Dynamic management of component properties at instance or class level is allowed through the functions `spw_setproperty`, `spw_getproperty` and `spw_getproperties`, where properties concern both attributes and methods. The `spw_call` is a library function generic method invocation. In particular, `c.f(x,y)` is called as `spw_call(c, "f", x, y)`. Additionally, the former style is also possible through the method dispatch adaptation technique implemented in Figure 12. More specifically, given a the unique name associated to component handle `c` during runtime, then `c.f(x,y)` is semantically equivalent to `components.a.f(x,y).

The extension API is briefed in Figure 14. Function `spw_registercomponent` registers a new component class to the system, identifiable by a unique class name, while `spw_unregistercomponent` removes a previously registered component class. The `spw_setcomponentmetadata` function allows the provision of class metadata, in particular a human readable name, a brief functionality description (documentation), the author (programmer) and version information. All metadata attributes are strings. The component metadata are displayed by the interactive introspector, as earlier shown under Figure 6.

Some of the supplied functions in the extension API support the linkage with the extensible User Interface template. More specifically, any function (of signature `void (void)`) of a Delta component class may be dynamically introduced as a User Interface.
command via the `spw_registerfunction`, `spw_registerslot` and `spw_registercommand` functions, as far as an external name (from the UI) and an internal function object (from the Delta script) are provided. As with `spw_registerfunction`, the `spw_registerslot` is also used for registering exported (public) component methods, while the method is automatically linked with a named signal (notification slots). Hence, slots are method entries directly connected to signals, automatically called when the respective signal is triggered, i.e. they are callbacks. Components may specify raised signals associated to globally unique identifiers and function signatures, with which respective slots should comply. A slot method may optionally accept as its first argument the component handle of the signal emitter, while the rest of its signature should directly match the respective signal’s signature. In this context, the `spw_registersignal` function installs new types of signals, through their signature and unique identifier, while `spw_signal` is the way to raise a signal.

Additionally, it should be noted that the User Interface commands are linked only to component classes, rather than to component instances. This technical decision is compliant to the way typical windowing applications handle multiple instantiations of specific component class, such as document editors: *in case of multiple component instances for a given class, command-invocation notifications due to user actions are propagated only to the component instance owning the focus*. Exactly the same policy is adopted in the Sparrow IDE. An example of an extension component built in the Delta language, showing how it is incorporated in the User Interface dynamically, is provided in Figure 15.

```javascript
function CountFileLOCs(script) {
    script = spw.getcomponent(script);
    local uri = script.GetURI();
    // Uniform resource identifier
    Count lines logic goes here, setting 'lines' local variable
    return lines;
}

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

classId = "LineCounter";
spw_registercomponent(classId); // Reserve entry for a new LineCounter component class
spw_setcomponentmetadata(
    classId, "Workspace Line Counter",
    "Count LOCs of all scripts in the current workspace",
    "Themistoklis Bourdenas <themis@ics.forth.gr>",
    "Alpha 1.0"
);
spw_registerimage("LineCounterImg", "resources/linecount.png");
spw_registercommand(
    classId, "CountLines", CountLOCs,
    "/{200}Tools/{100}Count Lines", // Reserving a UI command menu item
    spw.menu_tool, // Use the menu-based tool flags
    "LineCounterImg",
    "Display total LOCs for the current workspace"
);
```

*Figure 15: Implementing and installing a line counter component.*
In the Sparrow IDE, all component class methods should be defined with specific strongly-typed signatures. Since the Delta language is a dynamically typed language with implicit signatures (implied by the program logic), the latter need to be explicitly supplied upon function registration (see signature parameter in Figure 14), so that the Sparrow IDE component manager can perform strict signature conformance checking during runtime method invocations. Using such signatures, the internal implementation of `spw_call` is capable of performing strict signature checking upon method invocation. Besides the support of domain-specific component deployment and extensions, a powerful feature regarding meta-IDE functionality is the mechanism facilitating user-defined adaptations and configurations. For instance, the activation of domain-specific application development components (tools) may be managed either in a rule-based fashion or through on-demand configuration scripts. Application-oriented tools and editors, like animation and terrain editors in our specific case of Game Maker 1.0 development environment, are to be implemented as dynamically loaded extension components, activated by configuration scripts on demand. Individualized configurations as part of an integrated application development environment may reflect different roles in the development process, such as: animation definition, terrain construction, game character behavior scripting, etc. This particular policy has been extensively practiced in the development of the Sparrow IDE, since most of the basic components were built as component extensions, activated on demand by means of adaptation-oriented configurations.

![Expression Watches component](image)

**Figure 16:** Installing the *Expression Watches* component with its metadata.

Two representative examples of the way user-defined components can be introduced concern the debugging watch facility, and the adaptations management User Interface. As shown in Figure 16, the watch facilities of the source level debugger are entirely implemented using the Sparrow IDE itself as an extension component in the...
Delta language. The definitions to export runtime methods of the expression watches component are shown by the calls to `spw_registerfunction` library function, while signal handlers are installed through calls to the `spw_registerslot` function (see Figure 16, the respective method signatures are also marked). The ability to optionally track down the signal emitter in a signal handler is illustrated within the last call to `spw_registerslot` of Figure 16: first argument of the `slotListItemActivated` signal handler is defined as `Handle invoker`.

In Figure 17 we show how the adaptations management User Interface has been similarly implemented as a dynamic add-on in C++, by providing the respecting loading specifications for the component class. Loading dependencies, such as `AdaptationsManager` User Interface depending from the User Interface `Shell` component in Figure 17, imply the implicit pre-loading in case the required components are not active when loading of a component takes place at runtime.

```
<?xml version="1.0" encoding="utf-8"?>
<extension version="alpha"
  <component id="AdaptationsManager" type="DynamicLib" urc="/extensions/AdaptationsManager.dll">
    <dependency>Shell</dependency>
  </component>
  <Component id="ProfileRoot" type="DynamicLib" urc="/extensions/AdaptationsManager.dll">
    <dependency>ProfileManager</dependency>
  </component>
  <Component id="Profile" type="DynamicLib" urc="/extensions/AdaptationsManager.dll">
    <dependency>ProfileManager</dependency>
  </component>
  <Component id="BaseConfigResource" type="DynamicLib" urc="/extensions/AdaptationsManager.dll">
    <dependency>Profile</dependency>
  </component>
  <Component id="BaseScriptConfig" type="DynamicLib" urc="/extensions/AdaptationsManager.dll">
    <dependency>BaseConfigResource</dependency>
  </component>
  <Component id="ProfileConfig" type="DynamicLib" urc="/extensions/AdaptationsManager.dll">
    <dependency>BaseConfigResource</dependency>
  </component>
</extension>
```

**Figure 17:** Loading information for the adaptations User Interface.

### Summary and Conclusions

We have presented the Sparrow IDE supporting rationalized configurations and on-demand configurations, circular extensions, and management as a meta-IDE for application-oriented growth. Our primary design objective has been the provision of an open, extensible and customizable tool. Our implementation and test processes directly reflected this objective. We firstly introduced the component infrastructure, including all support for deployment, extension, dynamic loading, message dispatching, and the extensible User Interface framework - this constituted the basic Sparrow IDE, reflecting a *tabula rasa* concept. Then, we essentially started testing, since all the rest of the basic components were developed as either native or circular exten-
sions, with one additional goal: native extensions were to be transformed to circular extensions as soon as possible. While these ideas have been instantiated in the context of an IDE for a specific language, all reported facilities could be well realized in the context of any popular IDE for mainstream languages.

Finally, as a closing remark, we consider that future IDEs will be focused in providing more flexible and open functional primitives, rather than delivering ready-made closed-world solutions. In this context the emphasis will be likely put on enabling end-users introduce desirable extensions and customizations, reflecting their individual programming habits, as well as any particular future and emerging programming approaches. This will essentially allow IDEs to evolve rapidly during use, according to programmers’ intentions, knowledge and skills, becoming far more flexible and dynamic programming instruments.

References


