JazzScheme: Evolution of a Lisp-Based Development System

Guillaume Cartier Louis-Julien Guillemette
Auphelia Technologies Inc.
{gc,ljg}@auphelia.com

Abstract

This article introduces JazzScheme, a development system based on extending the Scheme programming language and the Gambit system. JazzScheme includes a module system, hygienic macros, object-oriented programming, a full-featured cross-platform application framework, a sophisticated programmable IDE and a build system that creates executable binaries for Mac OS X, Windows and Linux. JazzScheme has been used for more than 10 years to develop commercial software.

1. Introduction

Lisp has a long tradition of sophisticated programming environments entirely built in Lisp. This tradition can be traced as far back as the Lisp Machines [22] that even went to the extent of running on Lisp-dedicated hardware. At the time, those environments were a driving force in the industry, pushing the envelope of what a programming environment could do.

More recent Lisp environments include Emacs [9], Macintosh Common Lisp [7] (now Clozure CL [5]), Allegro CL [1], LispWorks [11], Cusp [6] and DrScheme [12] (now DrRacket [13]). Yet, few of those offer a complete solution to the following needs:

- being open-source
- being entirely built in their own language for fast evolution and complete integration
- being able to handle large scale enterprise development

In this article we introduce JazzScheme, a Lisp-based development system focused on enterprise development, which has been used for more than 10 years to develop commercial software.

JazzScheme is an open-source development system comprised of the Jazz platform and the Jedi IDE. The Jazz platform comes with a programming language that extends Scheme and Gambit [8], and that includes a module system, hygienic macros and object-oriented programming. The platform features a cross-platform application framework and a build system that creates executable binaries for Mac OS X, Windows and Linux. Jedi is a modern, programmable Lisp-based IDE with advanced features targeted at the Lisp family of languages.

This article starts with a personal account by the creator and main developer of JazzScheme, the first author, on the context of its birth and evolution. We then provide an overview of the Jazz platform and the Jedi IDE.

2010 Workshop on Scheme and Functional Programming

2. History and evolution

This section is written in the first person as it is a personal account of the history and evolution of JazzScheme by its creator.

The Little Lisper: Love at first sight

What really started this long adventure was a visit to the university library by a mathematics undergraduate student more than 20 years ago. At that time I already had a passion for programming but apart from the pure thrill of it, no language had really touched my mathematical sensibility. It all changed the day I discovered a tiny leaflet called The Little Lisper [18]. It was electric. Love at first sight! From that day, I knew I wanted to do everything necessary to be able to program and create elegant and complex software using that language. Many thanks to its authors! Amusingly, it would only be 20 years later that I would get to write my first pure Scheme line of code!

Roots

In the years that followed I ended up doing most of my programming in LeLisp [16], ZetaLisp [22] and Common Lisp [21]. Many JazzScheme concepts can be traced to that heritage:

- Multiple values
- Optional and keyword parameters
- Logical pathnames
- User extensible readtable
- Formatted output
- Sequences
- Restarts
- Object-oriented programming
- Metaclasses
- Generic functions
- Loop iteration macro (more precisely Jonathan Amsterdam's iterate macro)

Common Lisp After all those years of writing Common Lisp code, my dream was still to be able to program in Scheme for its purity and beautiful concepts. But Common Lisp offered so many features I needed, which pushed me into making many naive attempts to bridge the two worlds. Those attempts ended up deepening my understanding of the issues but still left me with the unsatisfying choice of having to choose between Scheme and Common Lisp.

Prisme In 1990 I graduated with a Master's Degree in mathematics and started looking for a job as a programmer. By chance, I met an old friend from the chess world, Renaud Nadeau, who had recently started his own company in Montreal, Micro-Intel, based on

Prisme, a Scheme-inspired language. Joining the company was appealing as it offered a dynamic work environment focused on the production of high-quality multimedia titles. On the other hand, Prisme, compared to Macintosh Common Lisp (MCL) [7], the development system I was using at the time, seemed primitive. In the end, the prospect of working with a dynamic team won me over and I joined Micro-Intel.

I then discovered that having complete access to the source code of the system had enormous benefits. After just a couple of weeks of intense hacking, I had added to Prisme most of my favorite tools from MCL.

I also discovered that I thoroughly enjoyed building real-life concrete applications. Highly graphical applications with real end users and real needs. This passion would be the guiding light during all the years that would eventually lead to the creation of JazzScheme, to have the best possible development system to build those applications.

Birth of "classic" Jazz

After working with Micro-Intel for 8 years, evolving Prisme, creating a complete IDE for it called Visual Prisme and writing many applications with their wonderful team of talented graphic artists, domain specialists and programmers, I wanted to learn what was at the time a complete mystery to me: the Information Technology (IT) world, e.g. systems programming for large corporations. I left Micro-Intel and became immersed in the world of databases, large-scale enterprise systems made of many subsystems, legacy code, distributed computing and languages such as Visual Basic and Java.

This is also the time, in 1998, when I started the Jazz project. I felt at the time that no other language than Lisp came close to having the potential to do what I wanted a development system to do. Many interesting Lisp systems were around but, unfortunately, open-source was still in its infancy and so they were all closed-source. The Prisme experience had taught me the incredible flexibility of having access to the source code of every part of a system.

Having no rights to Prisme, I could not reuse the result of all those years of work. But in the end, starting from a clean slate was the best thing that could have happened to Jazz.

Visual Basic I was working in Visual Basic at the time and using Visual Basic's IDE really made me aware of the productivity gains that can be achieved by using a feature-rich IDE to code and debug. I also discovered Visual Basic's GUI designer, which was one of the best available at the time. Its property-based approach would become the seeds of Jazz's component system.

C++-based At that stage, I made the first and most crucial design decision so far, that is to write the whole interpreter for the functional and object-oriented core of the language in C++. The decision was primarily based on my experience with Prisme, for which the interpreter was written in C++.

In retrospect, I believe it would have been better to layer the system in order to minimize the amount of code written in a foreign language, and probably write only the functional layer in C++, building the object-oriented system on top of it using macros. This design decision would cost me many months of hard refactoring work later on, when Jazz was ported from C++ to Gambit. On the other hand, being able to write the first version of the system really quickly by leveraging previous experience in similar systems was a great gain.

Windowing system One noteworthy design decision was to use Windows' common controls, even though their limited functionality was no secret. The decision was made for two reasons:

1. After years of using the sophisticated Lisp IDE characterizing Prisme, I wanted to shorten as much as possible the time needed to build a first version of the new IDE in order to be able to do all my development with it as soon as possible.
2. Even though I wanted to implement the new controls entirely in Jazz, I knew that implementing a complete windowing system in Lisp would put enormous performance pressure on the language, which would force me to implement parts of the language like an optional type system early, diverting work from the IDE.

In the end, it was a good decision even though a lot of code had to be rewritten.

Another one joins In 2004, Stéphane Le Cornec joined in as a part-time contributor to Jazz. This talented individual and strong believer in the expressivity of Lisp-based languages has made many contributions to JazzScheme since then.

Jazz becomes open-source

A couple of years later, around 2001, I met Marc Feeley, Gambit's author (we later discovered that we were both present at the 1990 ACM Conference on LISP and Functional Programming, in Nice, France, but didn't know each other). After many interesting exchanges, Marc suggested porting Jazz from its C++ base to Gambit. The idea fit perfectly with one of my dreams, i.e. to do without the C++ layer. Marc wrote a proof-of-concept implementation of the core concepts of Jazz in Gambit, and the performance tests were convincing enough that we deemed the project feasible. At that time, though, Gambit and Jazz were still closed-source, which seriously limited the possibilities for collaboration.

In 2006, I decided to make the move to open-source and Marc had already done a similar move for Gambit some time before. The stage was set to port Jazz from C++ to Gambit. To reflect the fact that Jazz would finally be a proper implementation of Scheme, it was renamed JazzScheme.

The porting begins The first obstacle at that point was that, try as I may, I couldn't get Gambit to build on Windows, so I decided to look for other Scheme systems. This was acceptable as it was a goal to make JazzScheme as portable as possible across major Scheme implementations. To make a long story short, during the first six months, JazzScheme was running on Chicken [4], Bigloo [3] and PLT Scheme (now Racket [13]) but not Gambit! At that time Marc sent me a prebuilt version of Gambit for Windows and I was finally able to start developing JazzScheme for Gambit, the system that I already liked a lot and have learned to love since then.

I would like to personally thank Marc Feeley for his unwavering support and availability all those years. He was always prompt to fix bugs, add missing features to Gambit, and was always available for intense brainstorming sessions on how to improve those needed features into great additions to Gambit.

The present version of JazzScheme is Gambit-dependent but the portable core design remains, so it should be possible to port JazzScheme to other major Scheme implementations with a moderate amount of work.

Scheme was just too great! At that point, rewriting the C++ kernel into Scheme made the code so simple and clear that almost everything I had ever wanted to add to the Jazz language but hadn't been able to due to the difficulties of coding in a low-level language as C++, I was then able to do. The language was progressing by leaps and bounds.

Unfortunately, JazzScheme ended up to be a radically different, incompatible language compared to the old Jazz, forcing not only the implementation of a new language but also the porting of the 3000 or so classes constituting the existing libraries.

Here is a partial list of the incompatible features that were added to the new language:

- R5RS [15] conformance
- A new module system
- A new object-oriented syntax enabling tighter integration with the functional layer

To make the porting effort even more difficult, we started porting JazzScheme's GUI from being Windows specific to Cairo and X11; the whole process took two years.

So we ended up having to:

- Port the language from C++ to Gambit
- Port the existing libraries from the old Jazz to the radically different JazzScheme
- Port all the UI code from being Windows specific to being multi-platform

Lots of fun!

Lisp's syntax saves the day What saved the project at that point was Lisp's syntax as data and Jedi's many refactoring tools. When a change couldn't be done with a search and replace, it could often be done thanks to Jedi's ability to run a textual macro at every found occurrence. If that didn't work either, I would then write some Jazz code that would be run at each found occurrence, analyze the Scheme expression and output the replacement in the text buffer.

$95\times$ ***slower*** The first working version of the Gambit-based JazzScheme turned out to be $95\times$ slower than the old C++-based Jazz. Even load time was abysmal. A rough projection showed that it would take forever for Jedi to load completely at that stage. A big part of the problem was due to the naive quick implementation of many core features, but even apart from that, the new language was still immensely slower.

Statprof comes to the rescue Fortunately, Gambit has a statistical profiling tool called statprof [19] written by Guillaume Germain.

How such a useful tool as statprof could be written in so little code is remarkable. It is a tribute to Gambit and Scheme's clean design around powerful concepts as continuations. Statprof leverages Gambit's interrupt-based architecture and continuations to implement a complete statistical profiler in only 50 lines of Gambit code!

Using statprof, it was easy to identify all the hotspots. Here is a partial list:

Functions to macros It turned out that function call overhead was too great to implement the equivalent of the C++ low-level virtual table dispatch. Fortunately, Gambit offers access to a low-level unchecked API using `##` functions like `##car`, `##cdr` and `##vector-ref`. Most of these functions get compiled into native Gambit Virtual Machine (GVM) [17] calls that get turned into simple C code themselves. For instance, a call to `##vector-ref` will end up generating an array indexing operator in C.

To harness this power safely, though, we created an abstract macro layer on top of it where you could decide at build time if the macros should call the safe functions or the low-level ones without having to modify any source code. Those macros are all prefixed by `%%`, for example `%%car`.

More precisely, JazzScheme's build system was designed to support multiple configurations where you can specify the safety level for each configuration:

- core: jazz will generate safe code for every call even internal implementation calls
- debug: jazz will generate safe user code
- release: jazz will generate unchecked code

C inlining of class-of Statprof also showed that optimizing `class-of` was critical. Unfortunately, optimizing `class-of` using only Scheme code was not possible. Because Jazz supports using Scheme native data types in an object-oriented fashion, the implementation of class-of was forced to use an inefficient `cond` dispatch:

```
(define (jazz.class-of-native expr)
  (cond ((%%object? expr)       (%%get-object-class expr))
        ((%%boolean? expr)      jazz.Boolean)
        ((%%char? expr)         jazz.Char)
        ((%%fixnum? expr)       jazz.Fixnum)
        ((%%flonum? expr)       jazz.Flonum)
        ((%%integer? expr)      jazz.Integer)
        ((%%rational? expr)     jazz.Rational)
        ((%%real? expr)         jazz.Real)
        ((%%complex? expr)      jazz.Complex)
        ((%%number? expr)       jazz.Number)
        ((%%null? expr)         jazz.Null)
        ((%%pair? expr)         jazz.Pair)
        ((%%string? expr)       jazz.String)
        ((%%vector? expr)       jazz.Vector)
        ...
        ))
```

Using Gambit's `##c-code` C inlining special-form and Marc's in-depth knowledge of Gambit's memory layout for objects, it was possible to rewrite `class-of` into the following efficient version:

```
(jazz.define-macro (%%c-class-of obj)
      `(or (\#\#c-code #<<end-of-c-code
{
  ___SCMOBJ obj = ___ARG1;
  if (___MEM_ALLOCATED(obj))
  {
    int subtype = (*___UNTAG(obj) & ___SMASK) >> ___HTB;
    if (subtype == ___sJAZZ)
        ___RESULT = ___VECTORREF(obj,0);
    else if (subtype == ___sSTRUCTURE)
        ___RESULT = ___FAL;
    else
        ___RESULT = ___BODY_AS(___ARG2,___tSUBTYPED)[subtype];
  }
  else if (___FIXNUMP(obj))
      ___RESULT = ___ARG3;
  else if (obj >= 0)
      ___RESULT = ___ARG4;
  else
      ___RESULT = ___BODY_AS(___ARG5,___tSUBTYPED)[___INT(___FA
}
end-of-c-code
  ,obj                    ;; ___ARG1
  jazz.subtypes           ;; ___ARG2
  jazz.Fixnum             ;; ___ARG3
  jazz.Char               ;; ___ARG4
  jazz.specialtypes       ;; ___ARG5
  )
          (jazz.structure-type ,obj)))
```

Gambit based kernel faster than the old C++ kernel In the end, Gambit performed above all expectations (except maybe Marc's!) enabling the conversion of 200,000+ lines of C++ code into about 15,000 lines of Scheme code and improving the general performance of JazzScheme by a factor of about 2.

The porting of such a large code base with so many needs also forced Gambit to evolve during those years, ironing out many bugs in the process.

If JazzScheme ever gets ported to other Scheme systems, it could end up being an interesting large-scale benchmark of all those systems.

Jazz as a macro over Scheme I would like to elaborate on how all of this was possible because of Lisp's ability to extend the language using macros, which has always been one of its greatest strengths.

Traditionally, a language is implemented using another lower-level target language. The implementer usually writes a compiler that generates code in this target language and sometimes goes through the trouble of creating an interpreter that can be used for rapid development. Both writing a compiler and an interpreter are complex tasks which require years of dedicated effort to attain a high level of maturity. Also, if for simplicity purposes the compiler's target language is higher level and accessed through function calls, the danger is that the overhead of the function calls in the compiled code can become prohibitive.

The new Jazz language completely does away with having to write a compiler and interpreter by being implemented entirely as a macro over Gambit. This enables complete reuse of all the efforts dedicated to Gambit over the years and can be done with no performance overhead. This was by and large the main reason why the Jazz language implementation went from 200,000+ lines of C++ code to about 15,000 lines of Scheme code that even implemented many new features not found in the old Jazz!

I now see Gambit with its minimalist design focusing on key systems, as a wonderful language creation toolkit. It is the authors' opinion that Gambit could be used to implement many other languages using the same approach, even languages outside the Lisp family.

Object-oriented approach One of the most difficult decisions in the design of JazzScheme has to be how to implement object-orientation. Having used Common Lisp for many years, I was familiar, of course, with CLOS [20] and generic functions. In fact, I found very attractive how generic functions unify the functional and object-oriented layers of Common Lisp. On the other hand, the old Jazz object-orientation being based around class encapsulation, I was also painfully aware of how class encapsulation, when used where natural, could help manage a large code base like the old Jazz's 3000+ classes.

So, after many unsuccessful attempts at finding a totally satisfying solution that would have the advantages of both approaches, I finally decided that JazzScheme would support both approaches and that class encapsulation would be used where natural, but that we would also be able to rely on generic functions for more complex patterns.

A call to an encapsulated method `foo` on an instance `x` is represented using a special `~` syntax:

```
(foo~ x)
```

This syntax was chosen to make it as close as possible to a function call. Internally, it is referred to as a dynamic dispatch as JazzScheme will dynamically determine the class of `x` on first call and cache the offset of the `foo` method in the class vtable for efficient dispatch. If the type inference system can determine the class of `x` at compile time, it will be used.

Declarative language Another important design decision was to make JazzScheme a declarative language.

In a production environment, Scheme's dynamic nature, where definitions are only known at run time, can hurt greatly as any reference to an undefined symbol will only be known at run time, when the program happens to run at that exact place.

JazzScheme was designed to have a declarative structure to solve that problem. The code walker resolves all symbols at walk time and reports any unresolved symbol at that time. We say walk time instead of the more usual compile time as JazzScheme code can end up being code walked in three different situations:

- when compiling,
- when loading an interpreted module,
- when doing a live evaluation.

The declarative version of Scheme's `define` is the `definition` special form, which is so unsettling to new JazzScheme users coming from the Scheme world. There is really nothing strange about it, it is just a declarative version of `define` whose access can be controlled using a modifier such as `private` or `public` as in:

```
(definition public (relate x y)
  (cond ((< x y) -1)
        ((> x y)  1)
        (else     0)))
```

JazzScheme also fully supports the more familiar approach of explicitly exporting functionality using an `export` special form as in:

```
(export relate)

(define (relate x y)
  (cond ((< x y) -1)
        ((> x y)  1)
        (else     0)))
```

As those two approaches have advantages and supporters, JazzScheme supports both.

Built entirely in Jazz

Once the porting to Scheme was completed, around 2008, a long-standing dream had finally been fulfilled, that is to have a complete Scheme development system written entirely in itself. Indeed, having a system written in itself has many advantages:

Development cycle The most obvious advantage is, of course, the fast development cycle made possible by the use of a high-level language and IDE. It is not only having access to high-level constructs but also having only one language to focus on, both for implementation and as a user.

In the end, it all boils down to rapid evolution of both the language and the IDE. For example, often when we see something which could be improved in Jedi, we just do it live inside the IDE itself, test, correct, test and commit without restarting the IDE. With this fast development cycle, it is not uncommon to see 20+ commits per day on the JazzScheme repository with few developers.

Can be run fully interpreted In terms of the development cycle, great efforts were dedicated to the development of JazzScheme to make sure everything could be run interpreted without having to go through the slow process of compiling. Even such low-level parts of the system as the module system, the code walker and even the kernel can all be run 100% interpreted. For instance, even when adding new features to the module system, we often just modify the code, test, modify, test, ... and only when everything works do we build the system, which makes for a very fast development cycle.

Debugging Another advantage from the switch to Gambit was having access to a high-level debugger. The contrast between the C++ world and the Gambit world was never as sharp as when facing a difficult crash to debug. Developing the old Jazz Windows UI was a painful process, full of crashes, trying to reproduce the problem in the C++ debugger, and then hunting down arcane C++ structures far from the user code. The first time the Gambit debugger popped

up instead of what would have been a crash in the old system, with a high-level view of the stack, display of frames, ... the bug was solved in minutes. What a contrast!

Nowadays it is rare to end up in the Gambit debugger as JazzScheme's remote debugger handles almost all cases. It still happens sometimes that an internal bug ends up crashing the remote debugger, but then Gambit is still there to catch the problem and offer a convenient debugging environment.

Openness to the community The aforementioned language, Prisme, only had an interpreter. Because of that, a large proportion of the code (even parts as high-level as the text editor) was written in C. This was always one of the sorest points for the team of developers. Not having easy access to the source code and seeing an opaque C frame in the debugger made their work a lot harder. It also stopped them from being able to contribute fixes. This was especially painful because at that time, they were in excellent position to debug the problem. That realization influenced greatly JazzScheme's design to make it a language that could be compiled efficiently. With the porting of the C++ kernel to Gambit, JazzScheme users now have access to 100% of the code used to implement the system.

This can have far reaching implications:

- Learning: New users get access to a vast library of high-quality code to learn.
- Contributing: Contributing is easy as there is no "other" language and development system to learn.
- Debugging: Having access to all source code can improve debugging greatly.
- Deployment: Deployment can be made more modular as the system does not have to include a large kernel. This is especially important when working on large-scale projects.

Live by your word Dissatisfaction is one of the greatest driving forces in development. But how can you be dissatisfied with your language or IDE if they are not the tools you're using to develop them, like Visual Basic being coded in C. Using JazzScheme and Jedi to develop JazzScheme is a great driving force behind its development. There is rarely a single day where we do not improve JazzScheme or Jedi in some way.

Tribute to Lisp Above all other factors, building everything in JazzScheme is I think the greatest tribute to this extraordinary language that is Lisp!

Emacs

Lets relate the influence which Emacs had on Jedi over the years.

Emacs is one of the greatest development environments available, especially for Lisp languages. As such, almost everyone who has come to work with Jedi over the years comes from an Emacs background. Over and over these individuals have forced Jedi to evolve to meet Emacs's high standards of Lisp editing. In its latest version, Jedi now supports almost all Emacs core features and bindings, but there is no doubt that the next programmer who starts using Jedi will find tons of Emacs features he'd like to be added! Many thanks to Emacs and its dedicated team of maintainers.

The present: Auphelia

Last year, at the Montreal Scheme Lisp User Group (MSLUG), I met Christian Perreault an open-minded entrepreneur who had been looking for more than 10 years for a new technology which would enable him to create the next generation of his Enterprise Resource Planning (ERP) software. Was it a match made in heaven? After many intense discussions and evaluations lasting well over a month, Christian finally decided to use JazzScheme for the ERP backend, but reserved his decision on the UI frontend between QT and JazzScheme. Since then, the decision has been made to use JazzScheme across the whole system both for the backend and the UI frontend.

Also, I always had the dream to set up a work environment which would attract talented individuals from the Lisp world to work together on fun and challenging projects, and ultimately show the world what a Lisp-based development system could do. With Auphelia [2] this dream is actually coming true! Here is a quick presentation of the talented individuals who have already collaborated with us in the context of Auphelia:

Marc Feeley Marc Feeley is the author of Gambit, the Scheme system which JazzScheme is built upon. Being dedicated to the evolution of Gambit, Marc hasn't contributed directly to JazzScheme but he is always a great source of information and insight in intense brainstorming sessions about difficult issues.

Alex Shinn Alex Shinn is the well-known author of the IrRegex library [10] implementing regular expressions in pure Scheme. He is also the author of many other useful Scheme libraries and also recognized for his deep understanding of the intricacies of hygiene in a functional language such as Scheme.

Alex ported his IrRegex library to JazzScheme and integrated it into Jedi. He also added hygienic macro support to the module system and to the language in general.

The team Apart from those part-time collaborators, Auphelia includes at the time of writing this article a team of five programmers working full-time on the project. From that team, one to sometimes up to three work full-time on evolving open-source JazzScheme to support the various needs of the project.

3. Overview of the Jazz platform

JazzScheme is a language and development system based on extending Scheme and the Gambit system. Here is a brief overview of Gambit and the Jazz platform.

3.1 Gambit

JazzScheme is entirely built using Gambit-C, a high-performance, state-of-the-art R5RS-compliant Scheme implementation. Gambit offers a rich library including an API for accessing the compiler and interpreter. It conforms to the IEEE Scheme standard and implements 16 of the Scheme Requests for Implementation (SRFI) [14].

Our experience working with Gambit has confirmed its high level of reliability. As extensive as our use of it was, very few bugs were found over the past three years, and the few ones we came across were promptly addressed by its maintainer.

Gambit has shown it has all the essential features to make it the ideal platform for implementing a development system like JazzScheme. The ability to load compiled or interpreted code interchangeably is key to the fast development cycle promoted by Jazz. Gambit's capability to report errors in a precise and configurable manner allowed us in the debugger to present the frames in a way which closely matches the Jazz source code, abstracting away the artifacts of the macro expansion of Jazz into Scheme.

Implementing a responsive GUI-based application like an IDE is demanding in terms of performance and Gambit was up to the challenge. In particular, Gambit's efficient cooperative thread system was key to implementing a smooth user experience in the IDE. Also, porting JazzScheme and the UI framework to Linux / X11 showed that Gambit's implementation of all those features was highly portable.

3.2 JazzScheme

JazzScheme is a development system based on extending Scheme which includes a module system, hygienic macros, object-oriented

programming, a full-featured cross-platform application framework, and a build system which creates executable binaries for Mac OS X, Windows and Linux.

JazzScheme's object-oriented system supports single-inheritance with multiple interfaces, similar to Java, generic multi-dispatch functions *à la* Common Lisp, and metaclasses.

From the start, JazzScheme was designed to support highly interactive development:

- JazzScheme supports run-time redefinition of functions, methods, classes, *etc.* In Jedi, pressing Ctrl-Enter will send the selected block of code to the currently focused process for evaluation.
- Interpreted and compiled code can be loaded interchangeably. The JazzScheme kernel will automatically load a compiled version when one is up-to-date and load the code interpreted otherwise. The build system compiles each unit into a loadable object (i.e. a dynamic/shared library). Alternatively, the build system is capable of linking multiple units into a single loadable library, thus improving application load time.

The Jazz platform is comprised of a rich set of libraries, including:

- a sophisticated component system,
- an extensive, cross-platform UI library,
- full access to Cairo 2D graphics,
- a Lisp-based markup language,
- regular expressions,
- database access,
- networking,
- remoting,
- a crash handler in case of unrecoverable exceptions

4. Overview of the Jedi IDE

Jedi is a modern, programmable Lisp-based IDE with advanced features. Jedi is written entirely in JazzScheme and is one of the most complex applications built with JazzScheme.

Jedi has a code editor which supports a number of languages. Although Jedi is at its best while editing Jazz code, it also supports other Lisp dialects (Scheme, obviously, and Common Lisp), as well as C/C++, Java, JavaScript, TEX and others. For Lisp languages, Jedi supports syntax highlighting, Emacs-style editing [9], source code tabulation, customizable symbol completion and much more.

Common Lisp users will be happy to know that Jedi is soon to implement Emacs' Swank protocol for remote debugging, making it a full-fledged Common Lisp IDE.

Jedi supports rich editing modes and functions (Section 4.1), and integrates a number of useful tools for interacting with Jazz processes such as a remote debugger (Section 4.2) and profiler (Section 4.3), as well as a number of reflection tools (Section 4.4).

4.1 Jedi basics

Workspaces Jedi's user interface is customizable through the concept of workspaces which define the structure of the UI components and determines which tools are presented to the user. Workspaces are groups of related windows, tools, *etc.*, that are activated together. Jedi includes a primary workspace for editing text, as well as a debugger workspace (shown in Figure 5). There is also a groupware workspace to compare and merge files and directories, and a designer workspace to design graphical user interfaces for Jazz applications. At the right-hand-side of the IDE's toolbar is a set of buttons used to switch between workspaces. Workspaces are specified in a declarative sub-language of Jazz which allows the user to conveniently customize the IDE by changing the containment structure and properties of tool panels, splitter windows, *etc.*

Projects and files Projects and their source files are displayed in the workbench, appearing in the left-most panel of the IDE. A project is an entity that Jedi can build and run, possibly under control of the debugger. Projects are workbench entities that contain source files and resources. For every project, Jedi will build a full cross-reference database (its catalog) of every source file in that project. Note that projects can contain source code from any language, and Jedi will only catalog the source files that it knows about.

Cross-references Jedi maintains a database of cross-references in the code. This is particularly useful for exploring code. In Jedi, by placing the caret on a particular symbol in the code you can:

- Go to the symbol's definition (by pressing F12). The definition is opened in the editor; if multiple definitions of the symbol are found (e.g. a method with the same name can be found in different classes), they are listed in the search results window, as shown in Figure 1.
- Find references to this symbol (by pressing Shift-F12). Again, if only one reference is found, this reference is opened in the editor, otherwise the references/call sites are listed in the search results window.

Editing code

In addition to the cross-reference database, Jedi offers a rich set of code navigation facilities, allowing the user to:

- Browse the code by chapters (where chapters and sections are indicated by comments in the source code) or by following the hierarchy of declarations.
- Navigate backward/forward in the browsing history.
- Browse the class hierarchy.
- Perform an incremental search. Jedi has extensive search-and-replace capabilities with regular expressions support and textual macro recording for custom replace actions (cf. Section 4.5).

Code evaluation Jedi has a number of features for editing Lisp code that can enhance programmer productivity. In particular, you can evaluate code by pressing Ctrl-Enter in the text, and the expression where your cursor is will be evaluated in the focused process. You can evaluate a method, and the effect is to update the method's definition in the run-time system. The next time the method will be called, the new definition will be applied.

Text manipulations Jedi has extra text editing features familiar to Emacs users, such as the clipboard ring. You can copy multiple values to the clipboard (with Ctrl-C, applied repeatedly). Alt-V cycles in the clipboard ring and pastes, while Ctrl-V is the normal paste operation, which pastes the value at the current position in the clipboard ring.

4.2 Debugger

Jedi has a remote debugger with full source-level information. An advantage of remote debugging is that you are debugging your application exactly as itself with all its features: windows, menus, connections, ports, threads, ... instead of simulating inside the IDE its various features.

The debugger reports exceptions occurring in the remote processes and can display detailed information about their execution stack including the state of all variables in all active frames. The user can browse the individual frames and evaluate expressions in their context, and the IDE will highlight call sites in the code. Jedi's

debugger workspace (Figure 5) is composed of four panels at the top of the IDE which show, respectively:

1. The list of processes connected to the debugger. By default the Jedi process is connected to its own debugger, so if an exception occurs in Jedi, it will be presented in the debugger. There is also a distinguished process called the focused process which will be used when you evaluate code with Ctrl-Enter.
2. The list of threads in the focused process, with an icon indicating the thread's state. You can restart a thread stopped in an exception by right-clicking the thread and selecting a restart such as "Resume event loop".
3. The list of frames in the execution stack of the selected thread, as well as any exception on which the thread is stopped. This panel will also propose all available restarts in that thread (similar to the concept of restart in Common Lisp) when displaying an exception or break point.
4. The variables in the selected frame and their values. The state of structured objects is presented in a tree-like fashion as this pane is an instance of the object explorer (cf. Section 4.4)

Process snapshots The state of a Jazz process can be saved to a snapshot file, which can later be loaded into Jedi's debugger. Jazz applications actually have a crash handler which generates a process snapshot in case an unrecoverable exception occurs. Process snapshots once loaded in the debugger are presented in the same manner as for live processes, the only limitation being that objects can only be explored to some user-controlled depth.

4.3 Profiler

Jedi supports a remote profiler that is controlled using start/stop buttons that activate the profiler in the focused process, and presents the profile results as shown in Figure 2. When selecting an entry in the results list, Jedi automatically shows the call site in the source code. The profile results shown were collected by statprof [19], a statistical profiler for Gambit. The profiler distributes the running time according to the top n frames of the execution stack, so that you can identify not only which functions were called most often, but also what function called them, to a user-controlled depth.

4.4 Reflection tools

View explorer In Jedi (or other Jazz applications), if you are curious about what a particular widget does or how it is implemented, you can quickly find out using the view explorer, which gives information about a graphical component such as its class and properties. When the view explorer is activated (by pressing F8), you drag the cursor over the views in a window to select a view. After a second, a tooltip displaying information on a particular view is popped, as shown in Figure 3. You can then also get information on the enclosing components in the hierarchy by pressing the up arrow which selects the parent component. This way you can quickly find out about the structure of a complex user interface window and browse its implementation.

Object inspector The inspector tool used in the debugger allows the user to inspect the state of any object of a debuggee process. The inspector presents a list of the slots and properties of an object with their associated values. Object slots bound to jazz objects are recursively shown as trees. Structured values such as lists and vectors are shown as trees with their individual components divided. Note that the inspector creates the tree in a lazy fashion, so as to even out response time and avoid excessive overhead in memory.

4.5 Search and replace

It is not uncommon that a symbol such as a class name or method needs to be changed across the entire code base of Jazz which consists of about 1500+ files of Jazz and Scheme code. To support tasks like these, Jedi offers many search and replace functionalities accessed using the widget shown in Figure 4. It supports many modes and functions to specify custom replace actions and control the scope of the search.

You can specify multiple search/replace pairs that will be applied simultaneously. The search string can be an arbitrary regular expression (when the "Regexp" mode is selected), and you can refer to parts of the matching expression in the replace string. Moreover you can specify custom replace actions by selecting the "Play recording" mode, in which case the textual macro will be applied with the search string as the current selection.

By default, the scope of the search is limited to the text displayed in the active window, but can be set to span all the Jazz or Scheme files registered in the workbench, or to all the files in a given directory and/or the files with a given extension. It is also possible to search for definitions or references in specific projects of the workbench; for instance, you can find all text-related UI classes in Jazz by selecting the project jazz.ui and entering Text as search key.

5. Conclusion

In conclusion, JazzScheme has evolved from a dream to be able to use Lisp in everyday work to create fun, complex and engaging software to a mature Lisp-based development system used to build industrial software such as an Enterprise Resource Planning (ERP) application.

It is the authors' hope that JazzScheme ends up playing a small part in advancing the awareness to this incredible gem called Lisp which Lispers have been using for more than 50 years now. Not by telling about Lisp but by making it possible to create complex high-quality software so easily and rapidly that the programming community will ultimately and naturally be drawn to it.

References

[1] Allegro Common Lisp. `http://www.franz.com/products/allegrocl/`.

[2] Auphelia Technologies. `http://www.auphelia.com/`.

[3] Bigloo homepage. `http://www-sop.inria.fr/mimosa/fp/Bigloo/`.

[4] The Chicken Wiki. `http://chicken.wiki.br/`.

[5] Clozure CL. `http://openmcl.clozure.com/`.

[6] CUSP, a Lisp plugin for Eclipse. `http://www.bitfauna.com/projects/cusp/cusp.htm`.

[7] Digitool, inc. http://www.digitool.com/.

[8] Gambit-C, a portable implementation of Scheme. `http://www.iro.umontreal.ca/~gambit/doc/gambit-c.html`.

[9] GNU Emacs - GNU Project - Free Software Foundation. `http://www.gnu.org/software/emacs/`.

[10] IrRegular Expressions. `http://synthcode.com/scheme/irregex/`.

[11] LispWorks. `http://www.lispworks.com/`.

[12] PLT Scheme. `http://www.plt-scheme.org/`.

[13] Racket. `http://www.racket-lang.org/`.

[14] Scheme Requests for Implementation. `http://srfi.schemers.org/`.

[15] N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson. Revised^5 report on the algorithmic language Scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.

[16] Jérome Chailloux, Mathieu Devin, and Jean-Marie Hullot. LELISP, a portable and efficient LISP system. In *LFP '84: Proceedings of the*

1984 ACM Symposium on LISP and functional programming, pages 113–122, New York, NY, USA, 1984. ACM.

[17] Marc Feeley and James S. Miller. A parallel virtual machine for efficient Scheme compilation. In *In Lisp and functional programming*, pages 119–130. ACM Press, 1990.

[18] Daniel P. Friedman and Matthias Felleisen. *The little LISPer (2nd ed.)*. SRA School Group, USA, 1986.

[19] Guillaume Germain. statprof. `http://www.iro.umontreal.ca/~germaing/statprof.html`.

[20] Sonya Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, 1988.

[21] Guy L. Steele. *Common LISP: The Language*. Digital Press, Bedford, MA, 2. edition, 1990.

[22] Daniel Weinreb and David Moon. The Lisp Machine manual. *SIGART Bull.*, (78):10–10, 1981.

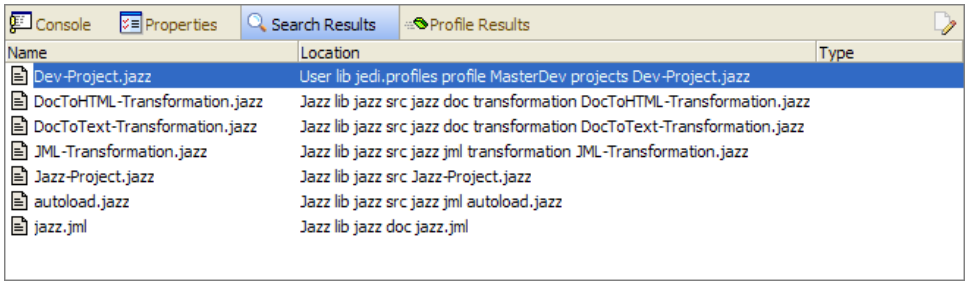

Figure 1. References to a program symbol shown in the Search Results pane.

Figure 2. Profiler results.

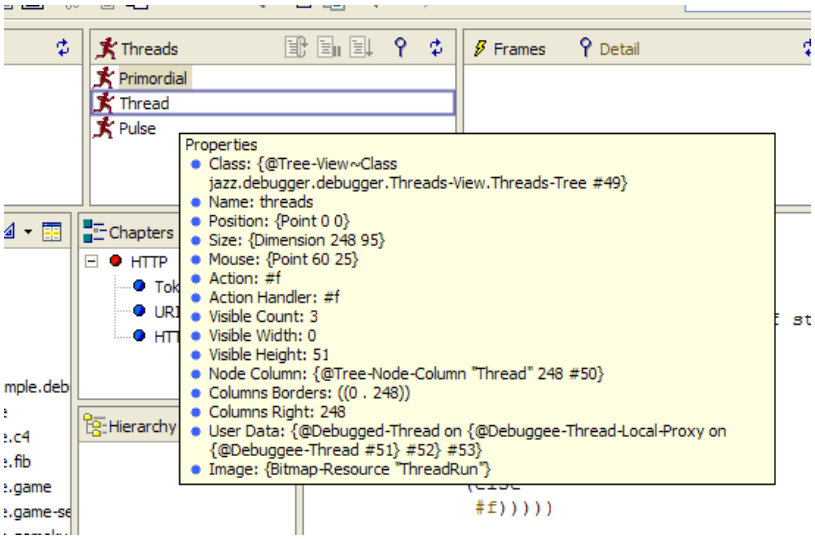

Figure 3. View explorer.

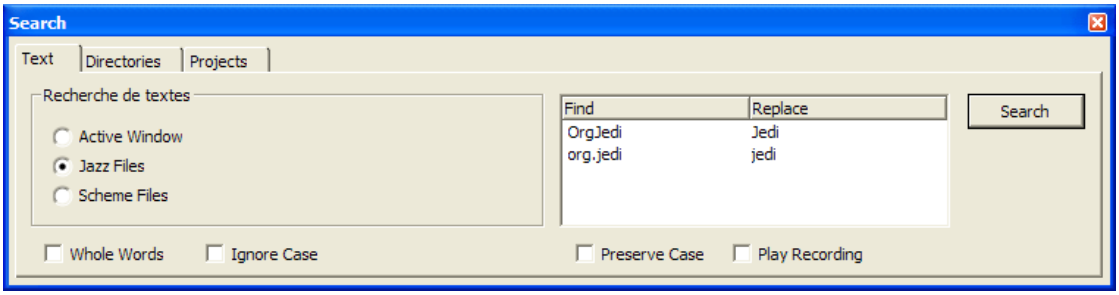

Figure 4. Search and replace.

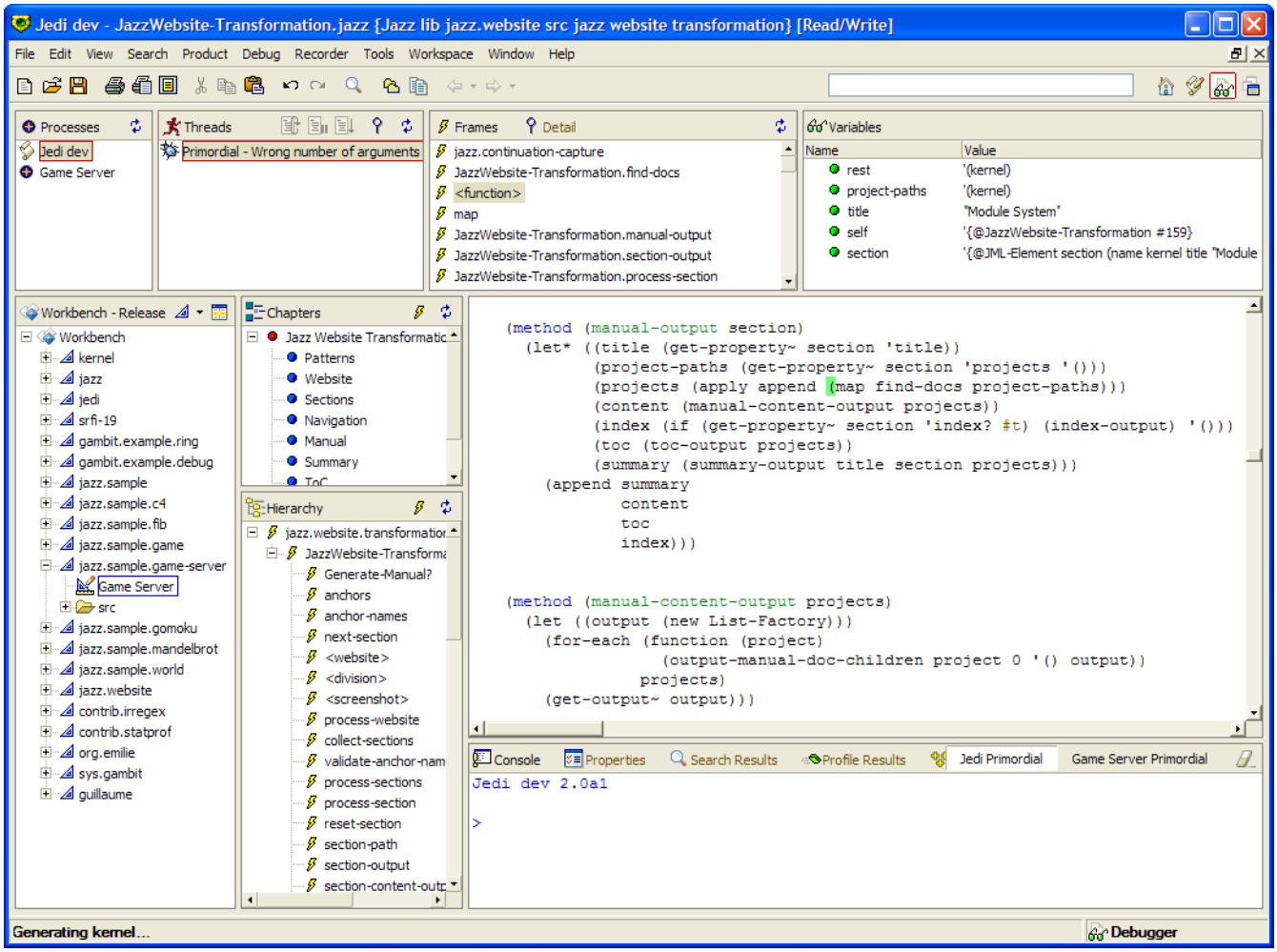

Figure 5. Debugger workspace.