Association for Computing Machinery
2 Penn Plaza, Suite 701
New York, NY 10121-0701

Ada LETTERS

Volume XXXIV Number 1 April 2014

Table of Contents

A Publication of SIGAda,
the ACM Special Interest Group on Ada

Volume XXXIV Number 1, April 2014

Table of Contents

A Publication of SIGAda,
the ACM Special Interest Group on Ada

For advertising information contact:
Advertising Department
2 Penn Plaza, Suite 701, New York, NY 10121-0701
Phone: (212) 869-7440; Fax (212) 869-0481

Is your organization recognized as an Ada supporter? Become a SIGAda INSTITUTIONAL SPONSOR! Benefits include having your organization's name and address listed in every issue of Ada Letters, two subscriptions to Ada Letters and member conference rates for all of your employees attending SIGAda events. To sign up, contact Rachael Barish, ACM Headquarters, 2 Penn Plaza, Suite 701, New York, NY 10121-0701, and email: MEETING@ACM.ORG, Phone: 212-626-0603.

Interested in reaching the Ada market? Please contact Jennifer Booher at Worldata (561) 393-8200 Ext. 131, email: platimer@worldata.com. Please make sure to ask for more information on ACM membership mailing lists and labels.

Ada Letters (ISSN 1094-3641) is published three times a year by the Association for Computing Machinery, 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA. The basic annual subscription price is $20.00 for ACM members.

POSTMASTER: Send change of address to Ada Letters:
ACM, 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA

From the Editor's Desk

Alok Srivastava

Welcome to this issue of ACM Ada Letters. In this issue you will find details on several remarkable Ada Gems and on Reusable Software Components by our veteran author Trudy Levine. In this issue you will find details on major Ada event, the 20th International Conference on Reliable Software Technologies Ada-Europe 2015 to be held from June 22-26, 2015 in Madrid, Spain.

Ada Letters is a great place to submit articles of your experiences with the language revision, tips on usage of the new language features, as well as to describe success stories using Ada. We'll look forward to your submission. You can submit either a MS Word or Adobe PDF file (with 1" margins and no page numbers) to our technical editor:

Pat Rogers, Ph.D.
AdaCore, 207 Charleston, Friendswood, TX 77546 (USA)
+1 281 648 3165, Rogers@AdaCore.Com

We look forward to hearing from you!

Alok Srivastava, Ph.D.
Technical Fellow, TASC Inc.
475 School St, SW; Washington, DC 20024 (USA)
+1 202 314 1419 Alok.Srivastava@TASC.Com

Editorial Policy (from Alok Srivastava, Managing Editor)

As the editor of ACM Ada Letters, I'd like to thank you for your continued support to ACM SIGAda, R&D in the areas of High Reliability and Safety Critical Software Development and encourage you to submit articles for publication. In addition, if there is some way we can make **ACM Ada Letters** more useful to you, please let me know. Note that Ada Letters is now on the web! See http://www.acm.org/sigada/ada_letters/index.html. The two newest issues are available only to SIGAda members. Older issues beginning March 2000 are available to all.

Now that Ada is standing on its own merits without the support of the DoD, lots of people and organizations have stepped up to provide new tools, mechanisms for compiler validation/assessment, and standards (especially ASIS). The Ada 2012 language version is fulfilling the market demand of robust safety and security elements and thereby generating a new enthusiasm into the software development. Ada Letters is a venue for you to share your successes and ideas with others in the Ada and specifically in High Reliability Safety Critical Software Development community. Be sure to take advantage of it so that we can all benefit from each other's learning and experience.

As some of the other ACM Special Interest Group periodicals have moved, Ada Letters also transitioned to a **tri-annual publication**. With exception of special issues, Ada Letters now is going to be published three times a year, with the exception of special issues. The revised schedules and submission deadlines are as follows:

Deadline	**Issue**	**Deadline**	**Issue**
June 1st, 2015	August, 2015	October 1st, 2015	December, 2015
February 1st, 2015	April, 2015	June 1st, 2015	August, 2015

Please send your article to Dr. Pat Rogers at rogers@adacore.com

Guidelines for Authors

Letters, announcements and book reviews should be sent directly to the Managing Editor and will normally appear in the next corresponding issue.

Proposed articles are to be submitted to the Technical Editor. Any article will be considered for publication, provided that topic is of interest to the SIGAda membership. Previously published articles are welcome, provided the previous publisher or copyright holder grants permission. In particular, keeping with the theme of recent SIGAda conferences, we are interested in submissions that demonstrate that "Ada Works." For example, a description of how Ada helped you with a particular project or a description of how to solve a task in Ada are suitable.

Although Ada Letters is not a refereed publication, acceptance is subject to the review and discretion of the Technical Editor. In order to appear in a particular issue, articles must be submitted far enough in advance of the deadline to allow for review/edit cycles. Backlogs may result in an article's being delayed for two or more issues. Contact the Managing Editor for information on the current publishing queue.

Articles should be submitted electronically in one of the following formats: MS Word (preferred) Postscript, or Adobe Acrobat. All submissions must be formatted for US Letter paper (8.5" x 11") with one inch margins on each side (for a total print area of 6.5" x 9") with no page numbers, headers or footers. Full justification of text is preferred, with proportional font (preferably Times New Roman, or equivalent) of no less than 10 points. Code insertions should be presented in a non-proportional font such as Courier.

The title should be centered, followed by author information (also centered). The author's name, organization name and address, telephone number, and e-mail address should be given. For previously published articles, please give an introductory statement (in a distinctive font) or a footnote on the first page identifying the previous publication. ACM is improving member services by creating an electronic library of all of its publications. Read the following for how this affects your submissions.

Notice to Contributing Authors to SIG Newsletters:

By submitting your article for distribution in this Special Interest Group publication, you hereby grant to ACM the following non-exclusive, perpetual, worldwide rights:

- to publish in print on condition of acceptance by the editor
- to digitize and post your article in the electronic version of this publication
- to include the article in the ACM Digital Library
- to allow users to copy and distribute the article for noncommercial, educational or research purposes

However, as a contributing author, you retain copyright to your article and ACM will make every effort to refer requests for commercial use directly to you.

Notice to Past Authors of ACM-Published Articles

ACM intends to create a complete electronic archive of all articles and/or other material previously published by ACM. If you have a work that has been previously published by ACM in any journal or conference proceedings prior to 1978, or any SIG Newsletter at any time, and you do NOT want this work to appear in the ACM Digital Library, please inform permissions@acm.org, stating the title of the work, the author(s), and where and when published.

Back Issues

Back issues of Ada Letters can be ordered at the price of $6.00 per issue for ACM or SIGAda members; and $9.00 per issue for non-ACM members. Information on availability, contact the ACM Order Department at 1-800-342-6626 or 410-528-4261. Checks and credit cards only are accepted and payment must be enclosed with the order. Specify volume and issue number as well as date of publication. Orders must be sent to:

ACM Order Department, P.O. Box 12114, Church Street Station, New York, NY 10257 or via FAX: 301-528-8550.

KEY CONTACTS

Technical Editor
Send your book reviews, letters, and articles to:

Pat Rogers
AdaCore
207 Charleston
Friendswood, TX 77546
+1-281-648 3165
Email: rogers@adacore.com

Managing Editor
Send announcements and short notices to:

Alok Srivastava
TASC Inc.
475 School Street, SW
Washington DC 20024
+1-202-314-1419
Email: Alok.Srivastava@tasc.com

Advertising
Send advertisements to:

William Kooney
Advertising/Sales Account Executive
2 Penn Plaza, Suite 701
New York, NY 10121-0701
Phone: +1-212-869-7440
Fax: +1-212-869-0481

Local SIGAda Matters
Send Local SIGAda related matters to:

Greg Gicca
Verocel
1849 Briland Street
Tarpon Springs, FL 34689, USA
Phone: +1-646-375-0734
Fax: +1-978-392-8501
Email: Gicca@Verocel.Com

Ada CASE and Design Language Developers Matrix
Send ADL and CASE product Info to:

Judy Kerner
The Aerospace Corporation
Mail Stop M8/117
P.O. Box 92957
Los Angeles, CA 90009
+1-310-336-3131
Email: kerner@aero.org

Ada Around the World
Send Foreign Ada organization info to:

Dirk Craeynest
c/o K.U.Leuven, Dept. of Computer Science,
Celestijnenlaan 200-A, B-3001 Leuven (Heverlee)
Belgium
Email: Dirk.Craeynest@cs.kuleuven.be

Reusable Software Components
Send info on reusable software to:

Trudy Levine
Computer Science Department
Fairleigh Dickinson University
Teaneck, NJ 07666
+1-201-692-2000
Email: levine@fdu.edu

SIGAda Working Group (WG) Chairs

See http://www.acm.org/sigada/ for most up-to-date information

Ada Application Programming Interfaces WG
Geoff Smith
Lightfleet Corporation
4800 NW Camas Meadows Drive
Camas, WA 98607
Phone: +1-503-816-1983
Fax: +1-360-816-5750
Email: gsmith@lightfleet.com

Ada Semantic Interface Specification WG
http://www.acm.org/sigada/wg/asiswg/asiswg.html
Bill Thomas
The MITRE Corp
7515 Colshire Drive
McLean, VA 22102-7508
Phone: +1-703-983-6159
Fax: +1-703-983-1339
Email: BThomas@MITRE.Org

Education WG
http://www.sigada.org/wg/eduwg/eduwg.html
Mike Feldman
420 N.W. 11th Ave., #915
Portland, OR 97209-2970
Email: MFeldman@seas.gwu.edu

Standards WG
Robert Dewar
73 5th Ave.
New York, NY 10003
Phone: +1-212-741-0957
Fax: +1-232-242-3722
Email: dewar@cs.nyu.edu

Ada Around the World
(National Ada Organizations)

From: http://www.ada-europe.org/members.html

Ada-Europe
Tullio Vardanega
University of Padua
Department of Pure and Applied Mathematics
Via Trieste 63
I-35121, Padova, Italy
Phone: +39-049-827-1359
Fax: +39-049-827-1444
E-mail: tullio.vardanega@math.unipd.it
http://www.ada-europe.org/

Ada-Belgium
Dirk Craeynest
C/o K.U.Leuven, Dept. of Computer Science, Celestijnenlaan 200-A, B-3001
Leuven (Heverlee), Belgium
Phone: +32-2-725 40 25
Fax : +32-2-725 40 12
E-mail: Dirk.Craeynest@cs.kuleuven.be
http://www.cs.kuleuven.be/~dirk/ada-belgium/

Ada in Denmark
Jørgen Bundgaard
E-mail: Info at Ada-DK.org
http://www.Ada-DK.org/

Ada-Deutschland
Peter Dencker, Steinackerstr. 25
D-76275 Ettlingen-Spessartt, Germany
E-mail: dencker@parasoft.de
http://www.ada-deutschland.de/

Ada-France
Association Ada-France
c/o Jérôme Hugues
Département Informatique et Réseau
École Nationale Supérieure des Télécommunications, 46, rue Barrault
75634 Paris Cedex 135, France
E-mail: bureau@ada-france.org
http://www.ada-france.org/

Ada Spain
J. Javier Gutiérrez
P.O. Box 50.403
E-28080 Madrid, Spain
Phone: +34-942-201394
Fax : +34-942-201402
E-mail: gutierjj@unican.es
http://www.adaspain.org/

Ada in Sweden
Rei Stråhle
Box Saab Systems
S:t Olofsgatan 9A
SE-753 21 Uppsala, Sweden
Phone: +46-73-437-7124
Fax : +46-85-808-7260
E-mail: Rei.Strahle@saabgroup.com
http://www.ada-i-sverige.se/

Ada in Switzerland
Ahlan Marriott
White Elephant GmbH
Postfach 327
CH-8450 Andelfingen, Switzerland
Phone: +41 52 624 2939
Fax : +41 52 624 2334
E-mail: ada@white-elephant.ch
http://www.ada-switzerland.org/

Italy
Contact: tullio.vardanega@math.unipd.it

Ada-Europe Secretariat
e-mail: secretariat@ada-europe.org

Gem #107: Preventing Deallocation for Reference-counted Types

Author: C.K.W. Grein, *Ada Magica*

Let's get started…

In Gem #97, a reference-counting pointer was presented, where a Get function returns an access to the data. This could be dangerous, since the caller might want to free the data (which should remain under control of the reference type). In this Gem, we present a method to prevent the misuse of the result of Get.

Let's repeat the relevant declarations:

```
  type Refcounted is abstract tagged private;
  type Refcounted_Access is access Refcounted'Class;

  type Ref is tagged private;  -- our smart pointer

  procedure Set (Self: in out Ref; Data: Refcounted'Class);
  function  Get (Self: Ref) return Refcounted_Access;

private

  type Ref is new Ada.Finalization.Controlled with record
    Data: Refcounted_Access;
  end record;
```

The function Get lets us retrieve and modify the accessed object. The problem with this function is that it compromises the safety of the pointer type Ref, in that a caller might copy the result access object and deallocate the accessed object:

```
  Copy: Refcounted_Access := Get (P);
  Free (Copy);
```

where Free is an appropriate instantiation of Unchecked_Deallocation.

To cure the situation, we no longer return a direct access to the data. Instead we define an accessor, a limited type with such an access as a discriminant, and let Get return an object of such a type:

```
  type Accessor (Data: access Refcounted'Class) is limited null record;
  function Get (Self: Ref) return Accessor;
```

Making the type limited prevents copying, and access discriminants are unchangeable. The discriminant also cannot be copied to a variable of type Refcounted_Access. The result is that the discriminant can be used only for reading and writing the object, but not for deallocation. Thus we have achieved our goal of making accesses safe.

A user might now declare some type derived from Refcounted and change the value of the accessed object like so:

```
declare
  type My_Refcount is new Refcounted with record
    I: Integer;
  end record;

  P: Ref;

begin
  Set (P, My_Refcount'(Refcounted with I => -10));
  My_Refcount (Get (P).Data.all).I := 42;
end;
```

This view conversion to My_Refcount will incur a tag check that will succeed in this example. In general, you have to know the type with which to view-convert in order to access the relevant components. An alternative is to declare a generic package like the following:

```
generic
  type T is private;
package Generic_Pointers is
  type Accessor (Data: access T) is limited private;
  type Smart_Pointer is private;
  procedure Set (Self: in out Smart_Pointer; Data: in T);
  function  Get (Self: Smart_Pointer) return Accessor;
private
  ... implementation not shown
end Generic_Pointers;
```

Instantiate with type Integer and the last line becomes instead:

```
Get (P).Data.all := 42;
```

So how do we implement the function Get? This is quite straightforward in Ada 2005, using a function returning a limited aggregate. (Note that in Ada 95, limited objects were returned by reference, whereas in Ada 2005 limited function results are built in place.)

```
function Get (Self: Ref) return Accessor is
begin
   return Accessor'(Data => Self.Data);
end Get;
```

Alas, we are not yet completely safe. To see this, we have to consider in detail the lifetime of the Accessor objects. In the example above, the lifetime of Get (P) ends with the statement and the accessor is finalized. That is, it ceases to exist (in Ada vernacular, the master of the object is the statement). So, tasking issues aside, nothing can happen to the accessed object (the integer in our example) as long as the accessor exists.

Now consider a variant of the above. Imagine we have a pointer P whose reference count is 1, and let's extend the accessor's lifetime:

```
declare
   A: Accessor renames Get (P);
begin
   Set (P, ...);  -- allocate a new object
   My_Refcount (A.Data.all).I := 42;  -- ?
end;  -- A's lifetime ends here
```

In this example, the master of the accessor is the block (and there are other ways to make the lifetime as long as one wishes). Now in the block, the pointer P is given a new object to access. Since we said that P was the only pointer to the old object, it's finalized with disastrous effect: A.Data is now a dangling pointer granting access to a nonexistent object until the end of the declare block.

(Note that this issue also existed in the original GNATCOLL.Refcount implementation.)

To cure the situation, we have to prevent the deallocation. That suggests increasing the reference count with the construction of an accessor and decreasing the count when the accessor is finalized again. The easiest way to accomplish this is to piggyback upon the properties of the smart pointer type:

```
type Accessor (Data: access Refcounted'Class) is limited record
  Hold: Ref;
end record;

function Get (Self: Ref) return Accessor is
begin
  return Accessor'(Data => Self.Data, Hold => Self);
end Get;
```

Incidentally, as a final note, the type Accessor should probably be declared as limited private, to avoid the possibility of clients constructing aggregates (which, by the way, would be quite useless).

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #108: Gprbuild and Configuration Files—Part 2

Author: Johannes Kanig, *AdaCore*

Let's get started…

In the first Gem of this series, we recalled that gprbuild is a tool to drive the build process of an Ada project automatically. The advantage of gprbuild over other tools is that it is not limited to Ada and is in fact highly configurable, namely by supplying a configuration file. In the previous Gem we gave an example configuration file that gives the necessary information to define a C compiler. Also recall the command line to invoke gprbuild with a custom configuration file:

```
   gprbuild --config=gcc.cgpr -P main.gpr
```

where main.gpr is the project file of a project containing C files to be compiled, and gcc.cgpr is the configuration file given in the previous Gem.

In this Gem, we show the overall structure of a configuration file and in particular describe a few attributes that were not explained in the previous Gem.

A configuration file consists of a configuration project, which is introduced as follows:

```
configuration project  is
    --  The content of the configuration project goes here
    --  ...
end ;
```

Such a configuration project contains a number of global attributes and a number of packages. The global attributes include:

```
  * the attribute Default_Language, which will be used by gprbuild if
the
    project does not define any programming language to be used;

  * a number of attributes related to libraries, shared libraries and
archives.
```

We have already seen a use of the Naming package, which contains attributes related to the naming schemes of the relevant languages. One can set the suffix for specification files and implementation files (appropriate for languages such as C and Ada), and one can also obtain the file name from the unit name for languages possessing the concept of a unit (currently Ada only). This can be achieved using the attributes Casing and Dot_Replacement.

```
package Naming is
   for Spec_Suffix ("Ada") use ".ads";
   for Body_Suffix ("Ada") use ".adb";
```

```
   for Casing ("Ada") use "lowercase";
   for Dot_Replacement use "-";
end Naming;
```

In the earlier Gem we introduced most of the possible attributes for the package Compiler:

```
package Compiler is
   for Driver ("Ada") use "/.../bin/gcc";
   for Required_Switches ("Ada") use ("-c", "-x", "ada", "-gnatA");
end Compiler;
```

There are also a number of options related to configuration files passed to the compiler, though we won't go into detail about those here.

Some compilers, such as gcc, are able to generate dependency information that can be exploited by gprbuild to offer a more efficient incremental recompilation process. The relevant attributes are part of the Compiler package, and for gcc the correct values are the following:

```
for Dependency_Switches ("C") use ("-Wp,-MD,");
for Dependency_Driver ("C") use ("gcc", "-E", "-Wp,-M", "");
```

The attribute Dependency_Switches gives the command-line arguments to cause the compiler to generate the dependencies and compile the source file at the same time, while the attribute Dependency_Driver specifies a command line that only generates the dependency file, without recompiling. In both cases, the compiler should generate a file with suffix ".d", containing lines of the form:

```
   b.o: b.c b.h a.h c.h
```

The filename before the colon is the name of an object file, and the list of files following the colon is the list of source files that this object file depends on. When any of these source files is more recent than the object file (gprbuild checks this by comparing the time stamp), the object file is regenerated.

The Compiler package also contains a list of options to specify the source search path information. The most basic variant is the attribute Include_Switches, specifying an option to be passed to the compiler:

```
     for Include_Switches ("C") use ("-I");
```

If there are many source directories, the command line can get too long. A better alternative is to use an environment variable which contains the search path, separated by colons:

```
     for Include_Path ("Ada") use "ADA_INCLUDE_PATH";
```

However, for large projects this may still not be enough; the attribute Include_Path_File defines an environment variable that can be used specify the name of a text file containing the list of search paths for sources:

```
for Include_Path_File ("Ada") use "ADA_PRJ_INCLUDE_FILE";
```

If several or all of the attributes related to source directories are present, gprbuild chooses according to the following preferences: Include_Path_File is used if it is defined, otherwise Include_Path, if it is defined, and Include_Switches is used as a last resort.

A related attribute is the attribute Mapping_File_Switches, which defines the compiler option to specify a mapping file. A mapping file is similar to the include path file, but it directly contains a mapping from unit names to file names, which is more efficient, especially in presence of remote source directories.

```
for Mapping_File_Switches ("Ada") use ("-gnatem=");
```

The package Binder specifies how to invoke the binder. As usual, one can specify the executable and mandatory options, but there are also two options similar to the ones for the source search path, allowing specification of the object search path.

```
package Binder is
   for Driver ("Ada") use "/.../gprbind";
   for Required_Switches ("Ada") use ("--prefix=");
   for Objects_Path ("Ada") use "ADA_OBJECTS_PATH";
   for Objects_Path_File ("Ada") use "ADA_PRJ_OBJECTS_FILE";
end Binder;
```

Finally, in the package Linker, one can specify the Driver (executable) for the linker, any required switches, and options to generate a map file:

```
package Linker is
   for Driver use "g++";
   for Map_File_Option use "-Wl,-Map,";
end Linker;
```

In the previous Gem we also mentioned that gprbuild is able to create a configuration file automatically, when none is provided, containing all necessary definitions for the preferred compiler for each language of the given project. This automatic generation is done by a tool called gprconfig, which obtains the necessary information from a set of XML files called the *knowledge-base*. In the next and final Gem of this series, we will learn how to use gprconfig, as well as how to add a compiler to the knowledge-base, so that it can be chosen automatically by gprconfig.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #109: Ada Plug-ins and Shared Libraries—Part 1

Author: Pascal Obry, *EDF R&D*

Let's get started…

This is the first part of a two-part Gem about Ada plug-ins. In this Gem, we discuss the way shared libraries are supported by GNAT and the interactions between shared libraries and the Ada run-time system. This is essential to fully understanding how to build Ada plug-ins, which are discussed in the second part.

In modern operating systems it is rare to have fully self-contained applications, that is, executables that require no external support from the operating system. Such applications occur mostly in the embedded world. In other domains, an application is built on top of other services that are accessible from libraries.

There are two basic kinds of libraries: static and dynamic. A static library is a simple container for object code that will be included in the final application's executable. This is done by the linker as the last step of creating an executable. That's why such libraries are called static (statically linked, with the library code embedded in the executable). Without static libraries we would need to link against many object files, so this simplifies linking. The other benefit of static libraries organized as archives of object files is to allow selective linking, so that only the necessary objects from the library are linked into the final executable.

A static library can be built with GNAT by using project files. For example:

```
library project MyLib is
    for Source_Dirs use ("src");
    for Object_Dir use "obj";
    for Library_Dir use "lib";
    for Library_Name use "mylib";
    for Library_Kind use "static";
end MyLib;
```

Dynamic libraries are quite a different story: they allow splitting the image of an executable into several pieces, some of which can be shared among different executables. Dynamic libraries offer two main advantages: 1) sharing code among executables, thus diminishing global memory usage, and 2) allowing modular maintainance of an application by having the capability of replacing a chunk of an application without having to regenerate or even interrupt the complete application. It is also important to note that dynamic libraries require direct support from the operating system.

Linking with a shared library is done in two steps. The first step occurs at link time, when the linker generates a table in the executable that contains the name of the shared library and all of the symbols that must be imported from the shared library. The second step occurs at load time using the dynamic linker.

The dynamic linker is part of the operating system and is in charge of finalizing the linking of the application. This step is quite complex, but let's summarize as follows. The dynamic linker:

1. Loads all shared libraries referenced by the executable.

2. Creates the executable-specific shared libraries' data sections (only code is shared between executables, not data).

3. Fills in the executable's corresponding shared library table with the actual addresses of the referenced symbols (variables and routines).

4. Calls the initialization routines (if any) defined in the just-loaded shared libraries.

At this point the application is ready to be run. Of course, shared libraries can reference other shared libraries; the dynamic linker is recursively doing the same job in this case. A very important point to understand is that whenever we have an Ada shared library used by an Ada application, both will be referencing the very same Ada run-time system. What is important in our context is that the elaboration is controlled by the executable and computed at bind time. This ensures proper Ada semantics as required by the standard.

A shared library can be built using a project file by setting the Library_Kind attribute to *relocatable*.

For plug-in support, the same piece of code may be loaded and unloaded several times and thus requires initialization to happen at load time and to be tied to the library itself; it cannot depend on the one-time global elaboration of the application.

GNAT has support for stand-alone shared libraries, and these are exactly the right kind of libraries to use in this context. A stand-alone library is a library that contains the necessary code to elaborate the Ada units that are included in the library and all the units it depends upon, recursively. Since many libraries may need to elaborate the same external units, the initialization code must be protected against multiple initializations of the same data objects.

Note that these libraries do not exactly conform to Ada semantics, since elaboration is supposed to happen only at program start and not in the middle when a plug-in is loaded. But such libraries are suitable for use with executables that are written in other languages or as plug-in libraries.

Declaring a library to be stand-alone is easy using GNAT Project files: it just requires adding an additional attribute in the library project file:

```
for Library_Interface use ("interface_unit1", "interface_unit2", ...);
```

This attribute declares the units that are visible from outside the library, and thus offers a hiding mechanism (for all units not in the interface), complementing the one offered by Ada: any unit not in the interface cannot be called from outside. This means, in particular, that changing the implementation of such units can be done without having to recompile, rebind, or relink the rest of the application.

In the second part of this series we will see how to set up an Ada plug-in framework based on stand-alone libraries.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #110: Ada Plug-ins and Shared Libraries—Part 2

Author: Pascal Obry, *EDF R&D*

Let's get started…

In the first part of this two-part series we saw that stand-alone shared libraries have the required properties to be used as dynamic plug-ins. In this Gem we explore how to load and unload code dynamically within an Ada application. In essence, the dynamically loaded code is compiled into a shared library, and when this shared library is modified it is reloaded automatically.

To accomplish this we need three modules:

- Registry -- A module handles plug-in loading, unloading, and service registration
- Computer -- A plug-in doing a simple computation using two integers and returning the result
- Main -- The main application that uses the computer plug-in

1.1.1 Registry

Each plug-in will inherit from a common type named Any. The associated access type Ref is used to record all loaded plug-ins in a hashed map:

```
package Plugins is
   type Any is abstract tagged null record;
   type Ref is access all Any'Class;
   --  A reference to any loaded plug-in
end Plugins;
```

Our Computer plug-in is described by:

```
package Plugins.Computer is

   Service_Name : constant String := "COMPUTER";
   --  Name of the service provided by this plug-in

   type Handle is abstract new Any with null record;
   type Ref is access all Handle'Class;

   function Call
     (H : not null access Handle; A, B : Integer) return Integer is
abstract;

end Plugins.Computer;
```

Note that this is only an abstract view that is shared by all the modules. This view describes all the routines supported by the plug-in. A concrete implementation of the computer plug-in will be given in the computer module.

The Registry spec is:

```
with Plugins;
package Registry is

   procedure Discover_Plugins;

   procedure Register
     (Service_Name : String; Handle : not null access
Plugins.Any'Class);

   procedure Unregister (Service_Name : String);

   function Get (Service_Name : String) return access
Plugins.Any'Class;

end Registry;
```

The Register, Unregister, and Get routines are trivial. The reference to the plug-in service is recorded in a hashed map by Register, removed by Unregister, and retrieved by Get. This part does not need further discussion.

The Discover_Plugins routine is the tricky one. Here is how it works. This routine scans the plug-ins directory for shared libraries prefixed by "libplugin_". If such a name is found, it is renamed by removing the "plugin" substring and loaded by the dynamic linker (see Shared_Lib.Load routine in the registry sources of the Gem's zip file). When the shared library is loaded, the elaboration code is used to register itself (that is, register the service name associated with the object reference) in the registry.

At this point the service is available and can be used by the main application.

Note that a map with all loaded shared libraries is kept. If a shared library is found to be loaded already, it is unloaded first. This calls the finalization code, which is used by the plug-in to unregister itself.

```
   procedure Discover_Plugins is

      function Plugin_Name (Name : String) return String is
         K : Integer := Strings.Fixed.Index (Name, "plugin_");
      begin
         return Name (Name'First .. K - 1) & Name (K + 7 .. Name'Last);
      end Plugin_Name;

      use Directories;
      use type Calendar.Time;

      S          : Search_Type;
      D          : Directory_Entry_Type;
      Only_Files : constant Filter_Type :=
                     (Ordinary_File => True, others => False);
      Any_Plugin : constant String :=
                     "libplugin_*." & Shared_Lib.File_Extension;
   begin
      Start_Search (S, "plugins/", Any_Plugin, Only_Files);
```

```
      while More_Entries (S) loop
         Get_Next_Entry (S, D);

         declare
            P     : Shared_Lib.Handle;
            Name  : constant String := Simple_Name (D);
            Fname : constant String := Full_Name (D);
            Pname : constant String := Plugin_Name (Fname);
         begin
            --  Proceed if plug-in file is older than 5 seconds (we do
not want to try
            --  loading a plug-in not yet fully compiled/linked).

            if Modification_Time (D) < Calendar.Clock - 5.0 then
               Text_IO.Put_Line ("Plug-in " & Name);

               if Loaded_Plugins.Contains (Pname) then
                  Text_IO.Put_Line ("... already loaded, unload now");
                  P := Loaded_Plugins.Element (Pname);
                  Shared_Lib.Unload (P);
               end if;

               --  Rename plug-in (first removing any existing plug-in)

               if Exists (Pname) then
                  Delete_File (Pname);
               end if;

               Rename (Fname, Pname);

               --  Load it

               P := Shared_Lib.Load (Pname);
               Loaded_Plugins.Include (Pname, P);
            end if;
         end;
      end loop;
   end Discover_Plugins;
```

The Shared_Lib spec comes with two bodies, one for Windows and one for GNU/Linux. The proper body is selected automatically.

Note that the renaming of the plug-in is required on Windows, as it is not possible to write to a shared library which is in use.

1.1.2 Computer

Here is the specification of the Computer module that is an implementation of the abstract type described above:

```
with Plugins.Computer;
package Computer is

   type Handle is new Plugins.Computer.Handle with null record;
```

```
   overriding function Call
     (H : not null access Handle; A, B : Integer) return Integer;

end Computer;
```

The body is straightforward. The Life_Controller is used to control the Computer object's life. The Initialize procedure (called when the plug-in is loaded) registers the service name and the Finalize procedure (called when the plug-in is unloaded) unregisters the service name.

```
with Ada.Finalization;
with Registry;
package body Computer is

   use Ada;

   type Life_Controller is new Finalization.Limited_Controlled with
null record;
   overriding procedure Initialize (LC : in out Life_Controller);
   overriding procedure Finalize (LC : in out Life_Controller);

   H : aliased Handle;

   overriding function Call
     (H : not null access Handle; A, B : Integer) return Integer is
   begin
      return A + B;
   end Call;

   overriding procedure Finalize (LC : in out Life_Controller) is
   begin
      Registry.Unregister (Plugins.Computer.Service_Name);
   end Finalize;

   overriding procedure Initialize (LC : in out Life_Controller) is
   begin
      Registry.Register (Plugins.Computer.Service_Name, H'Access);
   end Initialize;

   LC : Life_Controller;

end Computer;
```

1.1.3 Main

The main is the easy part. We just loop and once every second we discover plug-ins and call the Computer service if found.

```
with Ada.Text_IO;
with Plugins.Computer;
with Registry;
procedure Run is
```

```
   use Ada;
   use type Plugins.Ref;

   H      : Plugins.Ref;
   Result : Integer;

begin
   loop
      Text_IO.Put_Line ("loop...");
      Registry.Discover_Plugins;
      H := Plugins.Ref (Registry.Get (Plugins.Computer.Service_Name));
      if H /= null then
         Result := Plugins.Computer.Ref (H).Call (5, 7);
         Text_IO.Put_Line ("Result : " & Integer'Image (Result));
      end if;
      delay 1.0;
   end loop;
end Run;
```

To build and run the program, unpack the zip file attached to this Gem and execute the following commands:

$ gnat make -p -Pmain

$ gnat make -p -Pcomputer

$ ./run

Now, while the application is running, edit computer.adb and replace the addition in Call by a multiplication. Then recompile the computer plug-in:

$ gnat make -p -Pcomputer

After some time you'll see that the new plug-in code has been loaded automatically.

Note that an extended example illustrating dynamic loading and unloading is included as part of the GNAT examples.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #111: The Distributed Systems Annex, Part 5—Embedded Name Server

Author: Thomas Quinot, *AdaCore*

Let's get started…

In the first installment in this series of DSA Gems, we used an application managing a public bulletin board as a good example of a client-server design. We used the following configuration:

```
configuration Dist_App is
   pragma Starter (None);
   -- User starts each partition manually

   ServerP : Partition := (Bulletin_Board);
   --  RCI package Bulletin_Board is on partition ServerP

   ClientP : Partition := ();
   --  Partition ClientP has no RCI packages

   for ClientP'Termination use Local_Termination;
   --  No global termination

   procedure Display_Messages is in ServerP;
   --  Main subprogram of master partition

   procedure Post_Message;
   for ClientP'Main use Post_Message;
   --  Main subprogram of slave partition
end Dist_App;
```

and we obtained two executables, one for each partition (serverp and clientp), by issuing the following command:

po_gnatdist dist_app

With the above po_gnatdist configuration, running the application requires three steps:

- **Start po_cos_naming**, the PolyORB name server. On startup, an object reference is displayed, which must be passed to all partitions in the environment variable POLYORB_DSA_NAME_SERVICE or through a PolyORB configuration file.
- **Start serverp**, the server partition, which will register its RCI package (Bulletin_Board) with the name server.
- **Start clientp**, the client partition, which will query the name server for the location of the RCI.

Simplifying the process

To make this whole process easier, you can instead direct po_gnatdist to embed the name server within the main partition (serverp). This is achieved by saying:

```
pragma Name_Server (Embedded);
```

in the configuration file.

You can then start serverp without an external name server: it is now included within the partition. You still need to convey the location of the name server to clients. From within the server partition, this information can be retrieved by querying the PolyORB run-time parameters:

```
Put_Line ("ServerP started, embedded name server is at:");
Put_Line (PolyORB.Parameters.Get_Conf ("dsa", "name_service", ""));
```

This outputs a string of the form:

IOR:<...long series of hex digits>

which encodes all required information. You can then start the client partition by specifying this value in the POLYORB_DSA_NAME_SERVICE environment variable. In Bourne shell syntax, this translates to:

POLYORB_DSA_NAME_SERVICE=IOR:<...> ./clientp

Using the Windows cmd shell, this would be:

set POLYORB_DSA_NAME_SERVICE=IOR:<...>
client

Passing the name server information in a file

POLYORB_DSA_NAME_SERVICE can also indicate a file name, prefixed with the string "file:". So, if you modify serverp to output the name service reference to a file "ns.txt", you can start the client using

POLYORB_NAME_SERVICE=file:ns.txt ./clientp

Using a well-known port

Note: this requires the latest development version of PolyORB.

It is sometimes inconvenient to have to transport a string value or a file from the server to the client, and to have to update it each time the server is restarted. Using appropriate PolyORB run-time configuration directives, you can force the server to listen for network connections at a fixed location.

The following configuration forces the server to listen on port 8889:

```
[iiop]
polyorb.protocols.iiop.default_port=8889
```

Once this is set for the server, you can direct the client to that location by setting POLYORB_DSA_NAME_SERVICE to:

corbaloc:iiop:1.2@<hostname>:8889/_NameService

The included start_server and start_client scripts provide a demonstration of this facility.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #112: Lego Mindstorms Ada Environment—Part 1

Author: Pat Rogers, *AdaCore*

Let's get started…

The Lego Mindstorms processor has limited storage available, certainly not enough for a nontrivial application and a complete run-time library. Therefore, the GNAT tool-chain does not implement the full Ada language for this platform. A considerable number of language capabilities are still included, especially the Ravenscar tasking profile, but the entire language is not available.

For example, full exception semantics are not implemented. Programmers may raise user-defined and language-defined exceptions, and may handle them locally, but unhandled (and reraised) exceptions do not propagate dynamically up the call chain as they do in full Ada. Instead, an unhandled exception results in a call to a single global routine referred to as the "last chance handler," so called because it does not return to the application. Developers may define application-specific last chance handler replacements for the default implementation.

The default implementation shipped with this compiler simply shuts down the Mindstorms processor. However, the Ada interfaces define an alternative that displays the file name and line number corresponding to the location of the exception, using the LCD on the Mindstorms "brick." The routine also briefly buzzes the speaker to get the user's attention. It then goes into an infinite loop so that the user can note the location of the exception. This implementation is declared in the NXT.Last_Chance package, shown below.

```
with System;
package NXT.Last_Chance is

   procedure Last_Chance_Handler
     (Source_Location : System.Address;
      Line            : Integer);
   pragma Export (C, Last_Chance_Handler,
"__gnat_last_chance_handler");

end NXT.Last_Chance;
```

Note the pragma exporting the name as "__gnat_last_chance_handler", the routine name invoked by the run-time library when an unhandled exception is raised. Applications do not call this routine themselves, since it does not return to the caller and shuts down the system. Instead, they override the symbol and have the linker include their version of the routine in the executable so that the run-time library can call it if necessary. In the case of the Mindstorms interfaces, the user specifies the package NXT.Last_Chance in a with-clause, typically in the main program.

```
with NXT.Last_Chance;
```

```
...
procedure ...
```

You can see the source for the body of the procedure, indeed the sources for all the interfaces, in the “drivers” directory under your installation root. By default this would be:

C:\GNAT\2011\lib\mindstorms-nxt\drivers

Although not a very large or complicated procedure, in such a memory-constrained environment every byte may be precious, so you should take the size of the code into account. The procedure uses the audio interface to buzz the speaker, for example, and the LCD display package is also pulled in. If you were not otherwise using those packages (and their dependents), this could make a difference. However, there is no requirement for you to use this version of the last chance handler, so you can simply comment out or remove the with-clause if you don’t want the fancier handler and associated code included.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #113: Visitor Pattern in Ada

Author: Emmanuel Briot, *AdaCore*

Let's get started…

Imagine that you have a UML model and you want to generate code from it. A convenient approach is to have a "code generator" object, which has a set of subprograms to handle each kind of UML element (one that generates code for a class, one that generates code for an operation, etc.).

One way to implement this is by using a big series of **if** statements, of the form **if** Obj **in** CClass'Class **then**, which is rather inelegant and inefficient.

Another approach is to use discriminated types. A case statement on the discriminant is then efficient, and Ada will check that all discriminant values are covered. The problem is that then you would need to use case statements for all clients of the types in your application. Here, we prefer to use tagged types, to take advantage of Ada's OOP capabilities, so the case statement cannot be used.

Let's consider a specific example. Again, taking the UML example, assume we have the following types. These are only very roughly similar to the actual UML metamodel, but will be sufficient for our purposes. In practice, these types would be automatically generated from the description of the UML metamodel.

```
   type NamedElement is tagged private;
   type CClass is new NamedElement with private;
   type PPackage is new NamedElement with private;
```

In addition, a visitor class is declared, which will be overridden by the user code, for instance, to provide a code generator, a model checker, and so on:

```
   type Visitor is abstract tagged null record;

   procedure Visit_NamedElement
      (Self : in out Visitor; Obj : NamedElement'Class) is null;
   --  No parent type, do nothing

   procedure Visit_CClass (Self : in out Visitor; Obj : CClass'Class)
is
   begin
      --  In UML, a "Class" inherits from a "NamedElement".
      --  Concrete implementations of the visitor might want to work at
the
      --  "NamedElement" level (so that their code applies to both a
Class
      --  and a Package, for instance), rather than duplicate the work
for each
      --  child of NamedElement. The default implementation here is to
call the
```

```
      --  parent type's operation.

      Self.Visit_NamedElement (Obj);
   end Visit_Class;

   procedure Visit_PPackage (Self : in out Visitor; Obj :
PPackage'Class) is
   begin
      Self.Visit_NamedElement (Obj);
   end Visit_PPackage;
```

We then need to add one primitive Visit operation to each of the types created from the UML metamodel:

```
   procedure Visit (Self : NamedElement; V : in out Visitor'Class) is
   begin
      --  First dispatching was on "Self" (done by the compiler).
      --  Second dispatching is simulated here by calling the right
      --  primitive operation of V.

      V.Visit_NamedElement (Self);
   end Visit;

   overriding procedure Visit (Self : CClass; V : in out Visitor'Class)
is
   begin
      V.Visit_CClass (Self);
   end Visit;

   overriding procedure Visit (Self : PPackage; V : in out
Visitor'Class) is
   begin
      V.Visit_PPackage (Self);
   end Visit;
```

All of the code described above is completely systematic, and as such could and should be generated automatically as much as possible. The "Visit" primitive operations should never be overridden in user code in the usual case. On the other hand, the "Visit_..." primitives of the visitor itself should be overridden when it makes sense. The default implementation is provided just so the user has the choice at which level do to the overriding.

Now let's see what a code generator would look like. We'll assume that we are only interested, initially, in doing code generation for classes. Other types of elements (such as operations) will call the default implementation for their visitor (Visit_Operation, for instance), which then calls the visitor for its parent (Visit_NamedElement) and so on, until we end up calling a Visit operation with a null body. So nothing happens for those, and we don't need to deal with them explicitly.

The code would be something like the following:

```
   type CodeGen is new Visitor with private;
```

```
   overriding procedure Visit_CClass
      (Self : in out Codegen; Obj : CClass'Class) is
   begin
       ...;  --  Do some code generation
   end Visit_CClass;

   procedure Main is
      Gen : CodeGen;
   begin
      for Element in All_Model_Elements loop  --  Pseudo code
          Element.Visit (Gen);   --  Double dispatching
      end loop;
   end Main;
```

If we wanted to do model checking, we would create a type Model_Checker, derived from Visitor, that overrides some of the Visit_* operations. The body of Main would not change, except for the type of Gen.

When using this in practice, there are a few issues to resolve. For instance, the UML types need access to the Visitor type (because it appears as a parameter in their operations). But a visitor also needs to see the UML types for the same reason. One possibility is to put all the types in the same package. Another is to use "limited with" to give visibility on access types, and then pass an access to Visitor'Class as a parameter to Visit.

Here is a full example. This example must be compiled with the "-gnat05" switch since it uses Ada 2005 features such as the limited with clause and prefixed call notation.

```
with UML;         use UML;
with Visitors;    use Visitors;
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   type Code_Generator is new Visitor with null record;

   overriding procedure Visit_CClass
      (Self : in out Code_Generator; Obj : in out CClass'Class) is
   begin
      Put_Line ("Visiting CClass");
   end Visit_CClass;

   Tmp1 : NamedElement;
   Tmp2 : CClass;
   Tmp3 : PPackage;

   Gen  : aliased Code_Generator;

begin
   Tmp1.Visit (Gen'Access);  --  No output
   Tmp2.Visit (Gen'Access);  --  Outputs "Visiting CClass"
   Tmp3.Visit (Gen'Access);  --  No output
end Main;
```

```
limited with Visitors;
package UML is
   type NamedElement is tagged null record;
   procedure Visit
      (Self        : in out NamedElement;
       The_Visitor : access Visitors.Visitor'Class);

   type CClass is new NamedElement with null record;
   overriding procedure Visit
      (Self        : in out CClass;
       The_Visitor : access Visitors.Visitor'Class);

   type PPackage is new NamedElement with null record;
   overriding procedure Visit
      (Self        : in out PPackage;
       The_Visitor : access Visitors.Visitor'Class);
end UML;


with Visitors;  use Visitors;
package body UML is

   procedure Visit
      (Self        : in out NamedElement;
       The_Visitor : access Visitors.Visitor'Class) is
   begin
      The_Visitor.Visit_NamedElement (Self);
   end Visit;

   overriding procedure Visit
      (Self        : in out CClass;
       The_Visitor : access Visitors.Visitor'Class) is
   begin
      The_Visitor.Visit_CClass (Self);
   end Visit;

   overriding procedure Visit
      (Self        : in out PPackage;
       The_Visitor : access Visitors.Visitor'Class) is
   begin
      The_Visitor.Visit_PPackage (Self);
   end Visit;

end UML;


with UML;  use UML;

package Visitors is
   type Visitor is abstract tagged null record;

   procedure Visit_NamedElement
      (Self : in out Visitor; Obj : in out NamedElement'Class);
   procedure Visit_CClass
      (Self : in out Visitor; Obj : in out CClass'Class);
```

```
   procedure Visit_PPackage
      (Self : in out Visitor; Obj : in out PPackage'Class);

end Visitors;


package body Visitors is

   procedure Visit_NamedElement
      (Self : in out Visitor; Obj : in out NamedElement'Class) is
   begin
      null;
   end Visit_NamedElement;

   procedure Visit_CClass
      (Self : in out Visitor; Obj : in out CClass'Class) is
   begin
      Self.Visit_NamedElement (Obj);
   end Visit_CClass;

   procedure Visit_PPackage
      (Self : in out Visitor; Obj : in out PPackage'Class) is
   begin
      Self.Visit_NamedElement (Obj);
   end Visit_PPackage;

end Visitors;
```

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #114: Logging with GNATCOLL.Traces

Author: Emmanuel Briot, *AdaCore*

Let's get started…

When we write applications, we often add Put_Line statements to help debug the initial version of the code. Once that code works, we remove the Put_Line and move on to some other feature of the code. The problem is that when a bug occurs again in that part of the code (and especially when this is reported by a user and you can't debug on his machine), the output would still be useful, but it's no longer present in the executable.

The solution, of course, is to output the traces to some "log" file, and keep the traces forever in the code. The log file will eventually become very large if you have lots of traces, and finding information there might not be so easy. So, a better solution is to split the traces into various groups, and have a capability to display only some of the groups, so as to limit the amount of information to look at.

The GNAT Components Collection includes a package GNATCOLL.Traces that provides support for this. This package is used in various AdaCore products, in particular GPS.

The first thing to do is to initialize the module. The traces are configured via an external file, which by default is called ".gnatdebug", and is searched for in the current directory.

```
with GNATCOLL.Traces;   use GNATCOLL.Traces;
with User;
procedure Main is
begin
   Parse_Config_File;   --  parses default ./.gnatdebug
   User.Proc;
end Main;
```

The simplest way to compile this code is to use a project file. For instance:

```
with "gnatcoll";
project Default is
    for Main use ("main.adb");
end Default;

gprbuild -Pdefault.gpr
```

In the rest of the code we can create a stream. All log messages are sent to a stream, and you are free to create as many as you want. A log message will always be prefixed by the name of its stream, as a way to organize traces in the log file. Here is what the Ada code would look like:

```
package User is
   Stream1 : constant Trace_Handle := Create ("Stream1");
```

```
   procedure Proc is
   begin
      Trace (Stream1, "Some trace");
   end Proc;
end User;
```

If you compile this code and run it, there will in fact be nothing logged. That's because the trace streams are disabled by default. We thus need to create a configuration file. Its format is very simple. The following example demonstrates a number of its features:

```
+
> log
TRACE1=yes
TRACE2=no
TRACE3=yes > log2
DEBUG.COLORS=yes
```

The first line ("+") indicates that we would like to activate all the streams by default. So a file with only that line would already show the output in our Ada example.

The second line (">log") indicates where the output of the streams should go by default. If this isn't specified, the output goes to stdout. Otherwise, you can specify a file name. Advanced usage allow you to define other types of redirection. For instance, GNATCOLL comes by default with support for redirecting to syslog on Unix systems. It would be relatively simple to add support for custom outputs, like a socket, a database, etc.

The next three lines configure specific streams and show how to activate or disable them. The fifth line, in particular, redirects one of the streams to another log file.

The last line in the example activates one of the extra features of GNATCOLL.Traces. Activating "DEBUG.COLORS" will output the stream using different colors for the various information that is output on each line.

Other facilities are available. For instance, if you activate "DEBUG.ABSOLUTE_TIME", each line in the log file will include the time of the message. More powerfully, you can activate "DEBUG.STACK_TRACE" to get a stack trace for each message (note that the trace needs to be converted via addr2line). Another useful one is "DEBUG.COUNT", which will add the number of messages output so far in total and for the specific stream.

Here is what the log file will look like:

```
[STREAM1] 1/1 Some Trace (09:05:32.323)
[STREAM4] 1/2 Some Other Trace (09:05:33.125)
```

As we have seen in the example, text is output using the Trace function. There are two versions of it: the one we saw that is used to output a simple message, and another

version that takes an exception occurrence as a parameter and outputs information about that exception. There is also a procedure called Assert that will output an error message if a Condition is False. The error will be displayed in red if the colors were activated, thus making it easy to locate in the log file.

Preparing the text for a message might be expensive (for instance, if the text should contain the result of a function call, or requires a lot of string concatenation). For this, the recommended coding pattern is:

```
   if Active (Stream1) then
      Trace (Stream1, "Function result was " & Func(...));
   end if;
```

The log file might still be hard to read when it gets big. GNATCOLL.Traces provides a facility for indenting the traces. Here is a full example of code computing Fibonacci numbers recursively:

```
pragma Ada_05;
with Ada.Text_IO;       use Ada.Text_IO;
with GNATCOLL.Traces;   use GNATCOLL.Traces;

procedure Fibo is
   Me : constant Trace_Handle := Create ("FIBO");

   function Recurse (Num : Positive) return Positive is
      Result : Integer;
   begin
      if Num = 1 or else Num = 2 then
         return 1;
      end if;

      Increase_Indent (Me, "Computing" & Num'Img);
      Result := Recurse (Num - 1) + Recurse (Num - 2);
      Decrease_Indent (Me, "Done =>" & Result'Img);
      return Result;
   end Recurse;

begin
   Parse_Config_File;
   Put_Line (Recurse (5)'Img);
end Fibo;
```

The code should be compiled with a project file, as shown earlier. The current directory should also contain a file called ".gnatdebug" with a single line that contains "+". The output on stdout is:

```
[FIBO] 1/1 Computing 5
   [FIBO] 2/2 Computing 4
      [FIBO] 3/3 Computing 3
      [FIBO] 4/4 Done => 2
   [FIBO] 5/5 Done => 3
   [FIBO] 6/6 Computing 3
```

```
   [FIBO] 7/7 Done => 2
[FIBO] 8/8 Done => 5
 5
```

which shows the indentation that is increased for each recursive call.

GNATCOLL.Traces uses object-oriented code. It provides a number of hooks through which you can add your own extra data each time some line is output to the log file. For this, you should override the Pre_Decorator or Post_Decorator primitive operations.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #115: Lego Mindstorms Ada Environment — Part 2

Author: Pat Rogers, *AdaCore*

Let's get started…

The Mindstorms NXT "brick" contains both a 32-bit ARM processor and an 8-bit AVR processor. The ARM executes the application code, while the AVR is responsible for the lower-level device functionality, including controlling outputs, gathering sensor inputs, and even shutting down the brick itself.

The two processors coordinate by sending an unending stream of messages to each other. Upon initialization, the AVR begins sending messages containing information such as the current buttons pressed, if any, and the battery status, among other data. The ARM sends messages back to the AVR, for example to set power levels on outputs for motors or to command an analog-to-digital conversion to take place. The messages go back and forth continually at a fixed rate. In other words, they are not event-driven, although their content certainly reflects external events and internally generated commands. Look in the "drivers" directory for the package body of NXT.AVR if you want to see how these messages are sent and received. The one task declared in the entire API is located in that package body for that purpose.

The timing constraints on message handling are fixed by the hardware and are very strict. It is therefore possible for messages to be lost or garbled occasionally. A checksum is calculated to detect these errors, resulting in the problematic message being discarded.

The Ada API hides all these details behind high-level abstract data types for the sensors and motors, and procedural interfaces for the other lower-level facilities such as the A/D converter and discrete I/O capabilities of the ports. Nonetheless, application code must be written with some of the above architecture in mind. Specifically, incoming data are not available until the AVR has been initialized and has sent at least one message to the ARM. Although the AVR is initialized automatically by the Ada drivers, the application programmer must still await receipt of that message. For example, the current voltage of the battery is stored in a variable declared in the NXT.AVR package. The value of that variable is dependent upon arrival of a message from the AVR and is undefined beforehand. The accessor functions that decode the voltage representation simply process the variable as if the value is already defined. (There are ways to mitigate use of the undefined value, of course, but none are ideal.) Other values, such as the raw button readings, are also stored as variables that are set by the decoded AVR messages.

Therefore, the NXT.AVR package provides a procedure that awaits AVR initialization and initial message receipt. That procedure is named Await_Data_Available. A call to this procedure should typically be the first thing done by the application.

Another procedure is provided to power down the brick for those applications that are intended to complete (unlike, say, embedded control systems that only stop when external

power is removed). This is procedure Power_Down, also found in the NXT.AVR package declaration. The AVR is responsible for actually shutting down the power, so the procedure injects an appropriate message to the AVR into the message stream. However, as mentioned, messages can be lost or garbled, so the power-down request can be lost. As a result, it is best to use an infinite loop that calls Power_Down repeatedly. In effect, the loop "exits" when the AVR disconnects power to the brick. For example:

```
loop
   NXT.AVR.Power_Down;
   delay until Clock + Seconds (1);
end loop;
```

The absolute delay statement emulates a relative delay (as the Ravenscar subset doesn't include relative delays) by calling the Ada.Real_Time.Clock function and adding an interval to it. The duration of the interval is arbitrary and need not be one second.

Note that the NXT.AVR package also detects the situation in which the Power button on the brick is held down for an interval, and will shut down the brick automatically in response.

In summary, although the API goes to some lengths to hide the gory details of the ARM/AVR interactions, ramifications of the message-based architecture remain visible.

In the next Gem in this series we will explore the high-level abstract data types representing the sensors and motors.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #116: Ada and C++ Exceptions

Author: Quentin Ochem, *AdaCore*

Let's get started…

So we've decided to have some fun and are building an application that combines Ada and C++. So far, so good. We used the C++ to Ada binding generator to bind the C++ classes directly to Ada, and extend them. However, there's something that needs to be considered carefully –- what would happen if an exception were thrown/raised by C++ vs. Ada code? Let's try it out:

1.1.4 Step 1 — The C++ code

We're going to write some very simple C++ code, just two classes, one inheriting from the other, and overriding a "compute" primitive:

```
class COperation {
  public:
  virtual int compute (int a, int b);
  COperation ();
};

class CDivision : public COperation {
  public:
  virtual int compute (int a, int b);
  CDivision ();
};

int CDivision::compute (int a, int b) {
  if (b == 0) {
    throw new Problem ("Division by 0 in C++!");
  } else {
    return a / b;
  }
}
```

In the computation of CDivision, we'll raise a C++ exception if a divide-by-zero occurs. Let's add a second subprogram doing a dynamic dispatch:

```
void cpp_main (COperation & op) {
  try {
     cout << op.compute (1, 0);
  } catch (...) {
    cout << "Unknown exception caught, rethrowing..." << "\n";
    throw;
  }
}
```

This function calls the compute operation, catches the exception to display a message, and then rethrows it.

We're now going to bind this to Ada. Assuming all specifications are in a file base.hh, a simple call to gcc will do it:

```
g++ -fdump-ada-spec base.hh
```

1.1.5 Step 2 — Extending the C++ code in Ada

There is an implementation of CDivision in C++. After the binding phase, let's implement the same code in Ada:

```
type Ada_Division is new base_hh.Class_COperation.COperation with
     null record;

function Compute
  (this : access Ada_Division;
   a : int;
   b : int) return int is
begin
   if b = 0 then
      raise Constraint_Error with "Division by 0 in Ada!";
   end if;
   return a / b;
end Compute;
```

We now have three classes in the application. One root C++ class, one pure C++ child, and one mixed Ada/C++ child. Let's see how things how things work from there.

1.1.6 Step 3 — Catching a C++ exception in Ada

C++ exceptions do not have a known name once they reach the Ada world. They act just as if they were declared in the body of a package, so the only way to catch them is to use a general exception handler. Consider the following code:

```
   T_Cpp : aliased base_hh.Class_CDivision.CDivision;
   X : Interfaces.C.Int;
begin
   X :=  T_Cpp.compute (1, 0);
exception
   when others =>
      Put_Line ("[1] Exception caught...");
end;
```

This will print "[1] Exception caught…" on the screen.

1.1.7 Step 4 — Handling exceptions across languages

Now let's be a little bit more ambitious. We're going to call the cpp_main function with an object coming from Ada:

```
   T_Ada : aliased Base_Ada.Ada_Division;
```

```
begin
   base_hh.cpp_main (T_Ada'Access);
exception
   when Constraint_Error =>
      Put_Line ("[2] Constraint_Error caught...");
   when others =>
      Put_Line ("[2] Exception caught...");
end;
```

Now, looking back at the cpp_main implementation, we call compute with (1, 0) as the arguments. This will trigger an exception from the Ada implementation, which is going to be caught first by the C++ code, printing “Unknown exception caught, rethrowing…” on the screen. The same exception is then rethrown by the C++ side, ending up back in the Ada code above, printing “[2] Constraint_Error caught…”.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #117: Design Pattern: Overridable Class Attributes in Ada 2012

Author: Emmanuel Briot, *AdaCore*

Let's get started…

Most object-oriented programming languages provide a facility for declaring variables that are shared by all objects of a given class. In C++, these are called "static members" (and use the "static" keyword), and similarly Python has the notion of "class attributes".

Let's consider an example where this is useful. For instance, let's say we want to define the notion of a block of text that is generated by expanding a template (perhaps after we replace some parameters in that template, as can be done with AWS's templates parser, for instance). Once we have computed those parameters, we might want to generate multiple outputs (for instance HTML and CSV). Only the template needs to change, not the computation of the parameters.

Typically, such as in Python, the template could be implemented as a class attribute of the Text_Block class. We can then create templates that need the same information but have a different output simply by extending that class:

```
class Text_Block(object):
    template = "somefile.txt"
    def render (self):
        # ... compute some parameters
        # Then do template expansion
        print "processing %s" % self.__class__.template

class Html_Block(Text_Block):
    template = "otherfile.html"
```

In this example, we chose to use a class attribute rather than the usual instance attribute (`self.template`). This example comes from the implementation of GnatTracker: in the web server we create a new instance of Text_Block for every request we have to serve. For this, we use a registry that maps the URL to the class we need to create. It is thus easier to create a new instance without specifying the template name as a parameter, which would be required if the template name was stored in the instance. Another reason (though not really applicable here) is to save memory, which would be important in cases where there are thousands of instances of the class.

Of course, the approach proposed in this Gem is not the only way to solve the basic problem, but it serves as a nice example of one of the new Ada 2012 features.

C++, like Ada, does not provide a way to override a static class member, so it would use a similar solution as described below.

Since Ada has no notion of an overridable class attribute, we'll model it using a subprogram instead (the only way to get dispatching in Ada). The important point here is

that we want to be able to override the template name in child classes, so we cannot use a simple constant in the package spec or body.

```
type Text_Block is tagged null record;
function Template (Self : Text_Block) return String;
function Render (Self : Text_Block) return String;

function Template (Self : Text_Block) return String is
   pragma Unreferenced (Self);
begin
   return "file_name.txt";
end Template;
```

The parameter Self is only used for dispatching (so that children of Text_Block can override this function). Since we prefer to compile with "-gnatwu" to get a warning on unused entities, we indicate to the compiler that it is expected that Self is unreferenced.

We could make the function Template inlinable, which might be useful in a few cases (for instance if called from Render in a nondispatching call), but in general there will be no benefit because Template will be a dispatching call, which requires an indirect call and thus wouldn't benefit from inlining.

And that's it. We have the Ada equivalent of a Python class member.

But so far there is nothing new here, and this approach is rather heavy to write. For instance, the body of Render could contain code like:

```
pragma Ada95;

function Render (Self : Text_Block) return String is
   T : constant String := Template (Text_Block'Class (Self));
begin
   ..  prepare the parameters for template expansion
   ..  substitute in the template and return it
end Render;
```

Fortunately, Ada 2012 provides an easier way to write this, using the new feature of *expression functions*. Since Template is a function that returns a constant, we can declare that directly in the spec, and remove the body altogether. The spec will thus look like:

```
pragma Ada_2012;

type Text_Block is tagged null record;
function Template (Self : Text_Block) return String
   is ("filename.txt");
function Render (Self : Text_Block) return String;
```

This is a much lighter syntax, and much closer to how one would do it in Python (except we use a function instead of a variable to represent a class member). A child of Text_Block would override Template using the same notation:

```
type Html_Block is new Text_Block with null record;
overriding function Template (Self : Text_Block) return String
   is ("otherfile.html");
```

Compared to Python, this is in fact more powerful, because some of the children could provide a more complex body for Template, so we are not limited to using the value of a simple variable as in Python. In fact, we can do this in the spec itself, by using a *conditional expression* (another new feature of Ada 2012):

```
pragma Ada_2012;

type Text_Block is tagged null record;
function Template (Self : Text_Block) return String
    is (if Self.Blah then "filename.html" else "file2.json");
function Render (Self : Text_Block) return String;
```

Finally, we can also make the body of Render slightly more familiar (in terms of object-oriented notation) using the dotted notation introduced in Ada 2005:

```
function Render (Self : Text_Block) return String is
   T : constant String := Text_Block'Class (Self).Template;
begin
   ..  prepare the parameters for template expansion
   ..  substitute in the template and return it
end Render;
```

Now the call to Template looks closer to how it would appear in those languages that provide overridable class members. Some will argue that this doesn't look like a function call and thus is less readable, since we don't know that we are calling a function. This is a matter of taste, but at least we have the choice.

There is one thing we have lost, temporarily, in the declaration of Template. If we compile with -gnatwu, the compiler will complain that Self is unreferenced. There is currently no way to add a pragma Unreferenced within an expression function. This has generated a discussion here at AdaCore and the issue is not resolved yet. The current two proposals are either to always omit the unused parameter warning when a function has a single parameter and it controls dispatching (precisely to facilitate this class member pattern), or else to use an Ada 2012 aspect for this, as in the following:

```
function Template (Self : Text_Block) return String
   is ("filename.html")
with Unreferenced => Self;
```

Note also that the use of expression functions in this Gem requires a very recent version of GNAT: the expression function feature wasn't available in older versions, and the initial implementation had some limitations.

Related Source Code Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #118: File-System Portability Issues and GNATCOLL.VFS

Author: Emmanuel Briot, *AdaCore*

Let's get started…

One of the important issues to address when porting code from one system to another is that of file systems.

There are several aspects of the handling of file names that vary across file systems. For one, the file system might be case-sensitive or case-insensitive. This refers to whether casing of the name is relevant when accessing a file on the disk. In addition, the file system can be case-preserving or not. On some systems, file names are converted to upper case systematically when they are displayed (MS-DOS and VMS are in that category). Systems that do not preserve casing are always case-insensitive.

As a result, there are three categories of file systems: case-sensitive/case-preserving (most Unix file systems), case-insensitive/case-preserving (NTFS) and case-insensitive/case-destructive (FAT and VMS).

When running on a case-insensitive file system, applications should display file names with the same casing that the user used when creating them. However, many applications ported from Unix will simply convert all file names to lower case (to ensure uniqueness of file names internally) and thus have a display that is disturbing for the user. In practice, it has been our experience that file names should only be converted to lower case when comparing the names (for instance, to find out whether two names refer to the same file or when computing a hash). The rest of the time, the casing should always be preserved.

In truth, the introduction above is not quite precise enough: the attributes for casing really depend on the file system, not on the operating system (Windows, Linux,...). For instance, it is possible that your machine mounted a remote file system that has different properties (for instance, a Windows partition mounted on a Linux machine). Apple's OS-X is a special case here, in that its default is to be case-sensitive, but users can choose to make the file system case-insensitive. All of this shows that testing the system you are running on is not enough in practice. Unfortunately, we haven't found a good way to test the file system dynamically (not to mention that it would be very expensive, since for each file one would have to test for what file system it is on).

Another difficulty regarding file names is that of the character set in which they are encoded. When a file name only contains ASCII characters, there are in general no difficulties with manipulating the name. However, it is valid on most system to use accented characters in file names. But some file systems do not force an encoding, and just view the file name as a series of bytes, whose interpretation is left to applications (Windows Explorer, terminals, etc.) that will display the name. In general, those applications will take into account the user's locale for the display. Other file systems always interpret the file names as UTF-8. Again, for the application to get this exactly

right would require testing the file system for each file, rather than simply testing the system itself and assuming some defaults.

Another issue is symbolic links. On a lot of file systems, a file can be accessed through different paths when using symbolic links. Although these links have no impact when opening and reading the file, they make it more complicated to check whether two paths refer to the same physical file. This can be done by checking each component of the path to see whether it is a link, and if so, convert it to a normalized form. This computation can be expensive (especially on slow or remote file systems), so its result should be cached when possible.

The GNAT Components Collection (GNATCOLL) provides a useful package to abstract such aspects, namely GNATCOLL.VFS. This package provides several types that are used to manipulate files and their names:

```
type Filesystem_String is new String;
type Virtual_File is tagged private;
```

The first type above is intended as an initial replacement for the strings that are generally used to represent a file name. There is no conversion to or from Unicode. The intent is to remind users that the exact interpretation should not be a string that can be displayed as is, but a series of bytes that need to be interpreted in the context of a specific character set (most often UTF-8, but also ISO-8859-1 and variants).

Use of the second type involves a bigger change for most application: the idea is that it encapsulates and caches various information about a file and its name, and thus abstracts notions like case-sensitivity.

Let's consider some examples. We first need to get a representation for a file from the disk. For this, we can use one of the Create functions available in GNATCOLL. For instance,

```
declare
   F : Virtual_File;
begin
   F := Create ("/tmp/Foo.txt");
end;
```

If we pass F to some subprogram that should display it in a GUI, for instance, we expect the name to appear exactly as "Foo.txt", and not as an all-lower-case version "foo.txt", even on a case-insensitive file system. To get the name, we could, for example, write:

```
declare
   Name : constant Filesystem_String := F.Base_Name;
begin
   Put_Line (+Name);
end;
```

As noted earlier, the name should be considered as a series of bytes, the interpretation of which depends on your system. Most of the time, it is relatively safe to assume this is UTF-8. For such a case, GNATCOLL.VFS provides a "+" operator to convert the Filesystem_String to a String.

If we now create another instance of Virtual_File, we can test whether the two reference the same file. The result would be true on Windows, for instance, but not on Unix.

```
declare
   F2 : constant Virtual_File := Create ("/tmp/foo.txt");
begin
   if F = F2 then
      null;
   end if;
end;
```

On Unix, we could create a symbolic link from "/temp" to "/tmp". If we want our application to support symbolic links properly and recognize that "/temp/foo.txt" and "/tmp/foo.txt" are indeed the same file, we need to tell GNATCOLL that we are ready to pay the performance penalty, by calling:

```
GNATCOLL.VFS.Symbolic_Links_Support (True);
```

This support is turned off by default. When loading a big project in GPS for instance (with several thousand files) on a slow file system (ClearCase), not checking explicitly for symbolic links is at least an order of magnitude faster. That's why this is left as an explicit choice to the application.

However, GNATCOLL is clever enough to cache the symbolic resolution, as well as to normalize the file name. So if you reuse a Virtual_File several times, it will not need to perform the system calls again.

The API in GNATCOLL.VFS is much more extensive than what we have seen above, and provides ways to test whether we have a directory, whether the file is writable, read the contents of a file efficiently into memory, get the list of files in a directory, and even modify files. In each of these cases, GNATCOLL will make sure it uses the proper form of the file name when communicating with the system.

In fact, GNATCOLL.VFS also provides support for remote file systems (this is the basis of the remote mode in GPS), where network operations are performed transparently when you access a file, but this will be the subject of another Gem.

Converting an application to GNATCOLL.VFS is no small amount of work. But it provides a number of benefits in terms of portability and performance.

Related Source Code Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #119 : GDB Scripting—Part 1

Author: Jean-Charles Delay, *AdaCore*

Let's get started…

The GDB configuration file is located in the user's home directory:

```
${HOME}/.gdbinit.
```

When GDB starts, it sources this file -- if it exists -- meaning that it evaluates all the commands in the file, which can be any of the available CLI commands. At a basic level, this file can be used for simple configuration commands, such as selecting the default assembler format desired or changing the default input/output radix display style.

But GDB offers more: it can also read a macro-coding language that allows more powerful customizations.

This language follows the following format:

```
define _command_

  _code_
end
document _command_
_help text_
end
```

Commands defined like this are known as **user commands**. You can use and combine any standard GDB CLI commands in your user-defined macros.

The document section is important, since it is used by GDB to produce the output for the `help` command when associated with user-defined commands.

The GDB language also offers you a set of flow-control instructions and allows parameters. However, parameter manipulation is limited, because GDB only offers the following variables:

```
$argc

$arg0
$arg1
$arg2
...
```

However, note that GDB does not provide an array such as `$argv`.

Flow-control Instructions

GDB provides the "survival kit" for any language, meaning:

- The **set** statement
- The **if** control structure for conditions
- The **while** control structure for loops

The Set Statement

You can assign the result of an expression to an environment variable with `set`, for example:

```
set $VAR = 0

set $byte = *(unsigned char *)$arg0
```

The If Statement

```
if _expression_

  _statements_
else
  _statements_
end
```

The While Statement

```
while _expression_

  _statements_
end
```

Controlled Output

During the execution of a command file or a user-defined command, normal GDB output is suppressed; the only output that appears is what is explicitly printed by the commands in the definition.

GDB provides three commands for generating any desired output:

```
echo _text_
```

This command prints *_text_* including any nonprintable character escaped in a C-style string. No newline is printed unless you specify one using the `\n' character. You can also use escape sequences to output colors for a color terminal. For example, `\033[01;31'

prints the following characters in red, whereas `\033[0m' resets the colored output to default.

```
output _expression_
```

This prints the value of _expression_ and nothing but that value (no newline, no `$nn = ').

```
printf _string_, _expressions_...
```

This prints the values of the *_expressions_* under control of the format string *_string_*. The *_expressions_* may be either numbers or pointers.

For example, you can print two values in hex like this:

```
printf "foo, bar-foo = 0x%x, 0x%x\n", foo, bar-foo
```

Invoking the Shell

You can invoke a shell through the GDB command *shell*. If it exists, GDB uses the $SHELL environment variable to determine which shell to run. Otherwise it uses the default shell (`/bin/sh' on Unix systems, `COMMAND.COM' on MS-DOS, etc.).

Invoking make

Let's say that you compile your project using a Makefile and the `make` program. You may then want to use the GDB `shell` command to run `make`. GDB generally provides you with a `make` command of its own that executes the `make` program. For GNAT Pro Ada users though, the build program is `gprbuild`, which doesn't natively exist in GDB. However, you can define a command that will invoke it, for example:

```
define gprbuild

  if $argc == 0
      shell gprbuild
  end
  if $argc == 1
      shell gprbuild $arg0
  end
  if $argc > 1
      help gprbuild
  end
end
document gprbuild
Run the gprbuild program, optionally specifying the project name as
parameter.
Usage: gprbuild [project_name]
end
```

Radix Display Style

GDB allows you to change the default radix display style -- for both input and output -- allowing the following formats: octal, decimal, and hexadecimal.

You can always enter numbers in octal, decimal, or hexadecimal in GDB by the usual conventions: octal numbers begin with `0', decimal numbers end with `.', and hexadecimal numbers begin with `0x'.

However, numbers that begin with none of these are, by default, entered in base 10.

The radix commands are the following:

```
set input-radix _base_

set output-radix _base_
```

Caution: *base* must itself be specified either unambiguously or using the current default radix.

Therefore, you can select the hexadecimal style by default through the following commands:

```
set input-radix 0x10
set output-radix 0x10
```

Prompt Look

Changing the prompt look is also possible, and similar to changing your shell prompt. The following example makes your gdb prompt `gdb$' display in red:

```
set prompt \033[01;31mgdb$ \033[0m
```

Preventing GDB from Pausing during Long Output

You might have already experienced that when GDB needs to print more lines than your terminal height can display, it pauses each time the console is full. To prevent that and make GDB display all information at once, use the following settings:

```
set height 0
set width 0
```

Changing the Assembler Code Format

GDB supports different kinds of instruction set formats when disassembling a program via the `disassemble` or `x/i` commands.

You can set GDB to use either `intel` or `att` format. However, this command is only defined for the Intel x86 family:

```
set disassembly-flavor intel
set disassembly-flavor att
```

The default instruction set is `att` (the AT&T flavor used by default by Unix assemblers for x86-based targets).

Optional Messages

By default, GDB is cautious, and asks what sometimes seem to be a lot of questions to confirm certain commands. If you are willing to unflinchingly face the consequences of your own commands, you can disable this "feature":

```
set confirm off
```

Going Further

For a full list of GDB capabilities, see the GDB manual.

In a future Gem we will provide examples of GDB's more sophisticated scripting capabilities.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

REUSABLE SOFTWARE COMPONENTS

Trudy Levine
Fairleigh Dickinson University
Teaneck, NJ 07666
levine@fdu.edu
http://alpha.fdu.edu/~levine/reuse_course/columns

This column contains a listing of reusable software components, begun for Ada Letters in 1990. All information is obtained directly from parties affiliated with web sites hosting Ada components or from the sites themselves. As always, no recommendations or guarantees are implied. We appreciate comments, corrections, and suggestions from our readers.

Hard copies of Ada Letters (with green covers) going back to 1988 are available for reuse upon request.

~~~~~~

Ada-Belgium

One of the aims of the **Ada-Belgium** organization is to disseminate Ada-related information. So, in addition to the organization of seminars, workshops, etc., and the management of two mailing lists, it also has set up a site that enables everyone interested to consult and download a large variety of Ada-related information using a server in Belgium.

Key items include:

* **Conferences and events for the International Ada Community**
<http://www.cs.kuleuven.be/~dirk/ada-belgium/events/>
* **Ada job announcements, in or close to Belgium**
<http://www.cs.kuleuven.be/~dirk/ada-belgium/jobs/
* **A disk copy of the last version of the Ada and Software Engineering Library (ASE2, a 2 disk CD-ROM set).**
<ftp://ftp.cs.kuleuven.be/pub/Ada-Belgium/cdrom/index.html>
* A complete archive of the last public GNAT distribution that uses the GNAT Modified General Public License (3.15p).
<ftp://ftp.cs.kuleuven.be/pub/Ada-Belgium/mirrors/gnu-ada/>
* A directory with Free Ada Software provided by Belgian Ada users, including Rob Veenker's instructions for using native Ada application on an Android device:
<http://www.cs.kuleuven.be/~dirk/ada-belgium/software/>

The Ada-Belgium archive is primarily intended for the Belgian Ada community, but anyone interested in Ada is welcome to use it.

http://people.cs.kuleuven.be/~dirk.craeynest/ada-belgium/ (updated 2014)

~~~~~~

Ada Class Library

ACL is an object oriented library for Ada.Text search and replace. Scripting (small tool programs). CGI scripts. Execution of external programs (incl. I/O redirection). Garbage Collection. Extended Booch Components. CD-Recorder, Orto for command line parameter handling. An AdaCL release for Ada 2005 is included.

http://sourceforge.net/projects/adacl/ or **http://adacl.sourceforge.net (updated 2012)**

~~~~~~
~~~~~~

Ada Core

AdaCore provides open source tools and expertise for the development of mission-critical, safety-critical, and security-critical software. AdaCore's flagship products are the GNAT Pro and SPARK Pro development environments and the CodePeer automatic code reviewer and validator. The GNAT technology supports all four ISO standards of the Ada programming language - Ada 83, Ada 95, Ada 2005, as well as Ada 2012. GNAT Pro also comes with Frontline Support (provided by the developers of the toolset) and expert Ada consulting.

The GNAT technology includes:
- GNAT Programming Studio /GNATbench – Plug-In for Eclipse IDEs.
- Full Ada Compiler (Ada 83/Ada 95/Ada 2005/Ada 2012) Utilities for Analysis, Testing and Code Navigation, Visual Debugger, Libraries and Bindings, Runtime Profiles, and a range of mixed language solutions that allow programmers to write code in Ada, C, and C++ within a single development environment.
- GNAT Pro Safety-Critical and GNAT Pro High-Security products supporting safety and security standards.
- Support for over 70 native and cross platforms including Unix, Linux, Windows, RTOS (VxWorks, LynxOS, PikeOS, Android, iOS) and bare board support on ARM, PowerPC, x86, ERC32/LEON.
- Standalone additional technology includes:

Add-on technologies:
- GNATstack - Stack Analysis Tool, AWS - Web-Based Technologies, GtkAda - Intuitive GUI Builder and Extensive Widget Set, PolyORB - Middleware, ASIS-for-GNAT - Ada Semantic Analysis.
- GNATemulator - for fast target emulation on the host.
- GNATcoverage - for source and object code coverage.
- Traceability Study - for DO-178B/C level A source-to-object code traceability.

Standalone additional technology includes:
- SPARK Pro – code verification, based on information-flow analysis and theorem-proving. Includes support for SPARK 2014 (www.spark-2014.org).
- CodePeer - automatic code review and robustness validation.

Add-on technologies:
- GNATstack - Stack Analysis Tool, AWS - Web-Based Technologies, GtkAda - Intuitive GUI Builder and Extensive Widget Set, PolyORB - Middleware, ASIS-for-GNAT - Ada Semantic Analysis.
- GNATemulator - for fast target emulation on the host.
- GNATcoverage - for source and object code coverage.
- Traceability Study - for DO-178B/C level A source-to-object code traceability.

Learn how AdaCore technology can help your software development projects; contact info@adacore.com

The GNAT Academic Program (GAP) was created to help bring Ada to the forefront of university study. It includes a comprehensive toolset and support packages designed to give educators the tools they need to teach Ada.
Free Software developers and students can download GNAT GPL from
http://www.adacore.com/academia (updated 2014)
or contact: gap-contact@adacore.com
~~~~~~
~~~~~~

Ada Europe

Ada-Europe is an international organization, set up to promote the use of Ada. Ada-Europe represents European interests in Ada and Ada-related matters. Member organizations include:
Ada-Belgium, Ada-Denmark, Ada-Deutschland, Ada-France, Ada-Spain, Ada in Sweden, and Ada in Switzerland.
See: **http://www.ada-europe.org**
http://www.ada-europe.org/resources/online for Annotated Ada 2012 Language Reference Manual
updated 2014

~~~~~~

Ada IC

The Ada Information Clearinghouse has been providing free information about Ada and software engineering since 1990. The AdaIC maintains close contact with the Ada community in order to obtain the latest information on a variety of topics. Several blogs are maintained to continue conversion on Ada topics.

AdaIC links to the Ada Resource Association **http://www.adaic.org/community,** a consortium of Ada tool vendors and developers. The Ada Resource Organization sponsors the web site, www.ada2012.org, containing an overview of many new Ada 2012 features, as well as a list of Ada 2012 resources.

Visit Ada IC's website, **http://www.adaic.org** to see the latest in news, implementation guidelines, compilers and tools, reusable Ada code, education and training, Ada successes, and lessons learned by software developers. The site remains current with resources targeted for Ada 2012. An updated edition of the Ada 2012 Rationale is available at:
<http://www.ada-auth.org/standards/rationale12.html>

The Ada-wide search engine indexes all known Ada content (more than 76,000 pages according to Randy Brukhardt's last count). General search engines, such as Google, have too many references to the term "Ada" to make them practical for the purposes of the Ada community.

Please send any news you have on Ada to <news@adaic.org>. The Ada News of the AdaIC sometimes transmits press releases about the programming language to about 500 technical journalists and editors, as well as citing it on the AdaIC Website, as a free service to its users.

A comprehensive collection of Ada articles, reports, textbooks, videos, and CD-ROMS is available for browsing on-line through the AdaIC website. Users may access older components at the Virtual Library: **http://archive.adaic.com**
Ada 2012: http://www.adaic.org/ada-resources/standards/ada12
Reusable software components are available at
http://www.adaic.org/ada-resources/tools-libraries/ (updated May 2013)
~~~~~~

AJPO

The Ada Joint Project Office was closed on October 1998. For information on the AJPO see
http://sw-eng.falls-church.va.us/ajpofaq.html
http://sw-eng.falls-church.va.us/ajpo_databases/products_tools1.html
http://www.open-std.org/jtc1/sc22/wg9/ for the ISO home of Ada standards

~~~~~~
~~~~~~

Adalog

Adalog offers Ada utilities, Ada components, and Adapplets. These can be freely used and modified for any purpose, under the GMGPL license. Functions include Protection, Debugging, and OS_Services, among others.

The site also contains Adasubst/Adadep programs that are useful utilities for rearranging Ada programs, and AdaControl, a powerful utility for checking and enforcing style and coding rules.

AdaControl is a free (GMGPL) tool that detects the use of various kinds of constructs in Ada programs. Its first goal is to control proper usage of style or programming rules, but it can also be used as a powerful tool to search for use (or non-use) of various forms of programming styles or design patterns. Searched elements range from very simple, like the occurrence of certain entities, declarations, or statements, to very sophisticated, like verifying that certain programming patterns are being obeyed.

AdaControl supports most of Ada2005/2012 features. Since it is GMGPL, all of its parts can be reused for any purpose.

These programs are built on top of ASIS and include valuable packages providing higher level queries for ASIS (package Thick_Queries). For example, look for the function called "Full_Name_Image," which returns the unique name of any Identifier.

In addition, there is sc_timer, the Session Chair universal clock, which is very useful to those who have to chair a session, and a demo of GTK-Ada.

SEE: **http://www.adalog.fr**/ **(site updated 2014)**
http://www.adalog.fr/adalog2.htm
http://www.adalog.fr/compo1.htm for Ada components

~~~~~~

AdaPower

AdaPower.com is a repository of Ada information, links to resources, source code examples and packages for reuse. AdaPower.com can be divided into the following sections:

Articles and Links

Articles and Links to Ada Related Topics, Ada learning materials, and people in the Ada on-line community.

The Ada Source Code Treasury

Source code examples of using Ada and Ada related bindings and tools for both beginner and advanced students of Ada.

Packages for Reuse

An extensive repository of categorically arranged packages for download and links to packages and libraries available for reuse on the internet at
http://www.adapower.com/index.php?Command=Packages&Title=Packages+for+Reuse
http://www.adapower.com/index.php?Command=Class&ClassID=AdaLibs&Title=Ada+Libraries

see: **http://www.adapower.com/ (Site updated 2014)**

AdaPower's website has been entirely redone in Ada, using GRAW, a rapid agile web development framework, to maintain the repository of Ada information, links to resources, source code examples, and packages for reuse.

~~~~~~

Ada Structured Library Version 1.4

Ada Structured Library is a set of general containers and utilities. The library is licensed under the same license as GNAT (see GNU and FSF, below), which is GPL, but is modified to allow inclusion into a program without bringing the whole program under the GPL.

The utilities include some things lacking in Ada95, including:

* Abstract I/O - allows the I/O user and the I/O to be decoupled, so you can do file I/O, socket I/O, serial I/O, etc. by changing the I/O object the user references. Includes many functions of Ada.Text_IO.
* Calendar - Full-featured time and calendar manipulation.
* Telnet - A general telnet library implemented over sockets.
* Command processor - Does string tokenizing and command processing over Abstract I/O.
* A set of general-purpose containers, including Lists, Vectors, Trees, Graphs, and a Btreee, with lots of options.

See: http://adasl.sourceforge.net/
http://sourceforge.net/projects/adasl

~~~~~~

Ada-Switzerland

Ada-Switzerland is an association that promotes the use of the Ada programming language. In particular, it maintain links to resources and projects of the Ada Programming Language

See: **http://www.ada-switzerland.ch/** (updated 2014)
http://www.ada-switzerland.ch/resources.aspx

~~~~~~

Booch Components

The Ada 95 Booch Components began in late 1994 when David Weller ported Grady Booch's C++ components to Ada95. They have since been taken over by Simon Wright and Martin Krischik, and at this time, include implementations of bags, collections, dequeues, graphs, lists, maps, queues, rings, sets, stacks, and trees. These include definite and indefinite types, bounded and unbounded implementations, and dynamic and static storage allocations. Filtering and sorting operations are supported.

The Containers are compatible with Ada 95, Ada 2005, and Ada 2012 in GNAT 2012 mode. Backward compatibility is retained.

http://sourceforge.net/projects/booch95/ (Updated 2013)
http://booch95.sourceforge.net/documentation.html#the-containers
http://booch95.wiki.sourceforge.net/ComponentDocumentation#tocComponentDocumentation13

~~~~~~

Charles

Charles is a container and algorithms library for Ada, modeled on the C++ STL. Sequence containers (vectors, deques, and lists) store unordered elements, inserted at specified positions. Associative containers (sets, maps, multisets, and multimaps) order elements according to a key associated with each element; both sorted (tree-based) and hashed containers are provided. A separate iterator type associated with each container is used to visit container items and to effect direct modification of elements. Charles is flexible and efficient, and its design has been guided by the philosophy that a library should stay out of the programmer's way.

The Ada 2005 AI-302 reference implementation is located in the ai302 subdirectory:
http://charles.tigris.org/source/browse/charles/src/ai302/
See: **http://charles.tigris.org** **(Site updated 2005)**
~~~~~~

FREE SOFTWARE FOUNDATION

The Free Software Foundation is dedicated to eliminating restrictions on people's right to use, copy, modify, and redistribute computer programs. It promotes the development and use of free software and its documentation in all areas using computers. Specifically, it shepherds and helps fund a group of software developers of a complete, integrated software system named "GNU". ("GNU" is pronounced "guh-new" and stands for "GNU's Not Unix".) In 2013, GNU turned 30 years old.

The word "free" in "Free Software Foundation" refers to freedom, not price. You may or may not pay money to get GNU software, but regardless you have specific freedoms once you get it: the freedom to copy a program and give it away to your friends and co-workers; and the freedom to change a program as you wish, by having full access to source code. You can study the source and learn how such programs are written. You may then be able to port it, improve it, and share your changes with others. If you redistribute GNU software you may charge a distribution fee or give it away.

For the Free Software Definition, see: www.gnu.org/philosophy/free-sw.html

What is Copyleft?

The simplest way to make a program free is to put it in the public domain, uncopyrighted. But this permits proprietary modifications, denying others the freedom to use and freely redistribute improvements; it is contrary to the intent of increasing the total amount of free software. To prevent this, copyleft uses copyrights in a novel manner. Typically copyrights take away freedoms; copyleft preserves them. It is a legal instrument that requires those who pass on programs to include the rights to use, modify, and redistribute the code; the code and rights become legally inseparable.

The copyleft used by the GNU Project is made from the combination of a regular copyright notice and the "GNU General Public License." (www.gnu.org/copyleft/gpl.html) GPL is a copying license which basically says that you have the aforementioned freedoms. An alternate form, the "GNU Lesser General Public License" applies particularly to certain GNU libraries. This license permits linking the libraries into proprietary executables under certain conditions.

See www.gnu.org/copyleft/copyleft.html
www.gnu.org/licenses/licenses.html

GNAT is listed in the Free Software Directory, which catalogs useful free software that runs under free operating systems, particularly the GNU operating system and its GNU/Linux variants. The GNAT Technology includes the implementation of the ASIS standard (Ada Semantic Interface Specification), GtkAda to build portable and efficient GUIs in Ada, AWS (Ada Web Server) the framework to develop Web-based applications in Ada, the XML/Ada library to process XML streams in Ada, GLADE to develop distributed applications following the Ada Distributed Systems Annex standards, and PolyORB to develop distributed applications following the CORBA standard.

The GNAT GPL 2012 Edition, which is available free of charge from libre.adacore.com/, is licensed for Free Software development under the terms and conditions of the GNU General Public License (GPL).

For more information visit the following links:

GNAT Pro: www.adacore.com/gnatpro/
http://docs.adacore.com/gnatcoll-docs/ GNAT reusable components
http://directory.fsf.org/wiki/GNAT (site updated 2013)

Free Software Foundation, Inc.	+1 617 542 5942 x 23
51 Franklin Street, Fifth Floor	+1 617 542 2652 (fax)
Boston, MA 02110-1301	email: info@fsf.org

See: http://www.fsf.org (Site updated 2014)
http://www.gnu.org
https://directory.fsf.org/wiki/Ada_Components

~~~~~~
~~~~~~

GNAVI

GNAVI, the GNU Ada Visual Interface, is the open source alternative to visual software development languages like Delphi and Visual Basic. In addition to being fully Open Source under the GPL, the language foundation of GNAVI is the international standard of engineering, Ada. GNAVI for Windows offers comparable features to Delphi and Visual Basic, including use of Active X controls and the ability to interface with .NET and Java.

http://www.gnavi.org   (updated 2009)

Kazakov Objects

Dmitry Kazakov maintains a web site of free Ada components. The license is GM GPL, where appropriate. The library conforms to both Ada 95 and Ada 2005 language standards and includes:

1. Objects and handles (smart pointers)
2. Persistency
3. Sets and maps
4. Unbounded arrays
5. Unbounded arrays of pointers
6. Stacks
7. Pools
8. Doubly-linked networks
9. Graphs
10. Lock-free structures
11. Locking synchronization primitives
12. Parsers
13. Cryptography
14. Numerics
15. Miscellany
16. Networking
17. Packages
18. Installation
19. Changes log

See: www.dmitry-kazakov.de/ada/components.htm
www.download25.com/simple-components-for-ada-download.html (updated 2014)

Leake Components

Stephen Leake maintains the following Ada components:
Auto_Text_IO: automatically generates Text_IO packages for Ada packages
<http://stephe-leake.org/ada/auto_text_io.html>
Stephe's Ada Library: another entry in the Standard Ada Library sweepstakes
A large part of SAL provides math operations for kinematics and dynamics of masses in 3 dimensional space.Cartesian vectors, quaternions, orthonormal rotation matrices, moments of inertia, forces, acceleration, velocity are supported, in 3 and 6 degrees of freedom (translation and rotation). This library has been used for both robotics and satellite simulation.
<http://stephe-leake.org/ada/sal.html>
Emacs Ada mode: indentation, navigation, interface to GNAT tools for Emacs,
<http://stephe-leake.org/emacs/ada-mode/emacs-ada-mode.html>
An info version of the Ada 2005 and
> 2012 Reference Manuals and Annotated Ada Reference Manual in tar gzip format
OpenToken: LALR parser generator
<http://stephe-leake.org/ada/opentoken.html>
http://stephe-leake.org/   (updated 2014)

Matreshka

Matreshka is an Ada framework to help develop information systems. It includes:

- League --- a rich set of reusable core components to develop Ada applications. Its main purpose is to provide a high level abstraction tool for localization, internationalization and globalization of applications, as well as a portable interface to different operating systems. It contains many other useful features, among them advanced calendrical calculations, regular expressions, and JSON support to process and generate data in JSON format.
- XML processor --- provides the capability of manipulating XML streams and documents.
- Web framework
    - The FastCGI module assists with developing server side applications completely in Ada and using them with standard HTTP servers.
    - The SOAP module provides implementation of SOAP 1.2 protocol specification and assists in developing Web Services in Ada. This module includes implementation of standard security services:

      The WS-Security module provides SOAP message Security 1.1 (WS-Security 2004) and Web Service security: Username Token Profile 1.1.
    - WSDL to Ada translator
- SQL database access provides a simple generic API for accessing SQL databases. Supported databases include MYSQL, Oracle, Postgre SQL, SQLite 3, and Interbase/ Firebird
- Ada Modeling Framework provides implementation of OMG's Meta Object Facility (MOF) written completely in Ada. Extension modules are provided to assist in the analysis and modification of
    - UML models and their extensions: MOF Extensions, OCL models, UML Testing Profile
    - Diagram Definition

http://forge.ada-ru.org/matreshka **(updated 2014)**
Support of Matreshka is provided by QtAda

~~~~~~

USAFA

Professor Martin Carlisle at the US Air Force Academy continues to develop free software for use by the computer science community. Although previously known for tools specifically for Ada programmers (in particular A#, AdaGIDE, and RAPID), his more recent developments have targeted the computer science education and computer security audiences. The newest tools, RAPTOR and IRONSIDES, have Ada inside and are developed using AdaGIDE, GNAT, SPARK Ada and A#. RAPTOR is a flowchart-based programming environment useful for teaching introductory computer science and is taught in at least 22 countries.

IRONSIDES is a DNS server implemented in SPARK Ada using formal methods to prove the absence of many major categories of security vulnerabilities. (last updated 2014)

See: **http://ironsides.martincarlisle.com**
http://raptor.martincarlisle.com
http://adagide.martincarlisle.com
http://www.martincarlisle.com/ada_stuff.html
http://asharp.martincarlisle.com
http://rapid.martincarlisle.com

CONTACT: Martin C. Carlisle, Professor of Computer Science, US Air Force Academy
carlislem@acm.org
~~~~~~

FCRC'15
Federated Computing Research Conference
Oregon Convention Center, June 12 - 20, 2015
Portland, Oregon
http://fcrc.acm.org/

FCRC 2015 will be held in Portland Oregon from June 12-20. Chaired by Rajiv Gupta of UC Riverside, the conference will assemble a spectrum of affiliated research conferences into a week-long coordinated meeting. The technical program for each affiliated conference will be independently administered, with each responsible for its own meeting's structure, content and proceedings. To the extent facilities allow attendees are free to attend technical sessions of other affiliated conferences being held at the same time as their "home" conference. Conferences include:

- CRA-W 2015: Career Mentoring Workshop
- EC 2015: The 16th ACM Conference on Economics and Computation
- HPDC 2015: The 24th International Symposium on High-Performance Parallel and Distributed Computing
- CCC: Computational Complexity Conference
- ISCA 2015: The 42nd International Symposium on Computer Architecture
- ISMM 2015: ACM SIGPLAN International Symposium on Memory Management
- IWQoS 2015: IEEE/ACM International Symposium on Quality of Service
- LCTES 2015: ACM SIGPLAN/SIGBED International Conference on Languages, Compilers and Tools for Embedded Systems
- PLDI 2015: 36th ACM SIGPLAN Conference on Programming Language Design and Implementation
- SIGMETRICS 2015: International Conference on Measurement and modeling of Computer Systems
- SPAA 2015: ACM Symposium on Parallelism in Algorithms and Architectures
- STOC 2015: 47th ACM Symposium on Theory of Computing

Call for Papers

20th International Conference on Reliable Software Technologies — Ada-Europe 2015

22–26 June 2015, Madrid, Spain

Ada-Europe 2015

http://www.ada-europe.org/conference2015

Conference Chair

Alejandro Alonso
ETSIT-UPM
aalonso@dit.upm.es

Program co-Chairs

Juan A. de la Puente
ETSIT-UPM
jpuente@dit.upm.es

Tullio Vardanega
Università di Padova
tullio.vardanega@unipd.it

Tutorial Chair

Jorge Real
UPV
jorge@disca.upv.es

Exhibition Chair

Santiago Urueña
GMV
suruena@gmv.com

Industrial co-Chairs

Jørgen Bundgaard
Rambøll Danmark A/S
jogb@ramboll.dk

Ana Rodríguez
Silver Atena
ana.rodriguez@silver-atena.es

Publicity Chair

Dirk Craeynest
Ada-Belgium & KU Leuven
Dirk.Craeynest@cs.kuleuven.be

Local Chair

Juan Zamorano
ETSIINF-UPM
jzamora@fi.upm.es

"In cooperation" with ACM SIGAda, SIGBED, SIGPLAN, and with ARA

acm In-Cooperation

General Information

The 20th International Conference on Reliable Software Technologies – Ada-Europe 2015 will take place in Madrid, Spain. Following its traditional style, the conference will span a full week, including a three-day technical program and vendor exhibition from Tuesday to Thursday, along with parallel tutorials and workshops on Monday and Friday.

Schedule

11 January 2015	Submission of regular papers, tutorial and workshop proposals
25 January 2015	Submission of industrial presentation proposals
1 March 2015	Notification of acceptance to all authors
29 March 2015	Camera-ready version of regular papers required
12 April 2015	Industrial presentation abstracts required
17 May 2015	Tutorial and workshop materials required

Topics

The conference has over the years become a leading international forum for providers, practitioners and researchers in reliable software technologies. The conference presentations will illustrate current work in the theory and practice of the design, development and maintenance of long-lived, high-quality software systems for a challenging variety of application domains. The program will allow ample time for keynotes, Q&A sessions and discussions, and social events. Participants include practitioners and researchers representing industry, academia and government organizations active in the promotion and development of reliable software technologies.

Topics of interest to this edition of the conference include but are not limited to:

- **Multicore and Manycore Programming**: Predictable Programming Approaches for Multicore and Manycore Systems, Parallel Programming Models, Scheduling Analysis Techniques.
- **Real-Time and Embedded Systems**: Real-Time Scheduling, Design Methods and Techniques, Architecture Modelling, HW/SW Co-Design, Reliability and Performance Analysis.
- **Mixed-Criticality Systems:** Scheduling methods, Mixed-Criticality Architectures, Design Methods, Analysis Methods.
- **Theory and Practice of High-Integrity Systems**: Medium to Large-Scale Distribution, Fault Tolerance, Security, Reliability, Trust and Safety, Languages Vulnerabilities.
- **Software Architectures**: Design Patterns, Frameworks, Architecture-Centred Development, Component-based Design and Development.
- **Methods and Techniques for Software Development and Maintenance**: Requirements Engineering, Model-driven Architecture and Engineering, Formal Methods, Re-engineering and Reverse Engineering, Reuse, Software Management Issues, Compilers, Libraries, Support Tools.
- **Software Quality**: Quality Management and Assurance, Risk Analysis, Program Analysis, Verification, Validation, Testing of Software Systems.
- **Mainstream and Emerging Applications**: Manufacturing, Robotics, Avionics, Space, Health Care, Transportation, Cloud Environments, Smart Energy systems, Serious Games, etc.
- **Experience Reports in Reliable System Development**: Case Studies and Comparative Assessments, Management Approaches, Qualitative and Quantitative Metrics.
- **Experiences with Ada and its Future**: Reviews of the Ada 2012 new language features, implementation and use issues, positioning in the market and in the software engineering curriculum, lessons learned on Ada Education and Training Activities with bearing on any of the conference topics.

Program Committee

Mario Aldea, Universidad de Cantabria, Spain
Ted Baker, NSF, USA
Johann Blieberger, Technische Universität Wien, Austria
Bernd Burgstaller, Yonsei University, Korea
Alan Burns, University of York, UK
Maryline Chetto, University of Nantes, France
Juan A. de la Puente, Universidad Politécnica de Madrid, Spain
Laurent George, ECE Paris, France
Michael González Harbour, Universidad de Cantabria, Spain
J. Javier Gutiérrez, Universidad de Cantabria, Spain
Jérôme Hugues, ISAE, France
Hubert Keller, Institut für Angewandte Informatik, Germany
Albert Llemosí, Universitat de les Illes Balears, Spain
Franco Mazzanti, ISTI-CNR, Italy
Stephen Michell, Maurya Software, Canada
Jürgen Mottok, Regensburg University of Applied Sciences, Germany
Laurent Pautet, Telecom ParisTech, France
Luís Miguel Pinho, CISTER/ISEP, Portugal
Erhard Plödereder, Universität Stuttgart, Germany
Jorge Real, Universitat Politècnica de València, Spain
José Ruiz, AdaCore, France
Sergio Sáez, Universitat Politècnica de Valencia, Spain
Amund Skavhaug, NTNU, Norway
Tucker Taft, AdaCore, USA
Theodor Tempelmeier, University of Applied Sciences Rosenheim, Germany
Elena Troubitsyna, Åbo Akademi University, Finland
Santiago Urueña, GMV, Spain
Tullio Vardanega, Università di Padova, Italy

Industrial Committee

Jørgen Bundgaard, Rambøll Danmark A/S
Ana Rodríguez, Silver Atena, Spain
Dirk Craeynest, Ada-Europe & KU Leuven, Belgium
Jacob Sparre Andersen, JSA Consulting, Denmark
Jean-Loup Terraillon, ESA
Paolo Panaroni, Intecs, Italy
Paul Parkinson, Wind RIver, UK
Peter Dencker, ETAS GmbH, Germany
Rod White, MBDA, UK
Steen Palm, Terma, Denmark
Ahlan Marriott, White Elephant, Switzerland
Ian Broster, Rapita Systems, UK
Ismael Lafoz, Airbus Military, Spain
Jean-Pierre Rosen, Adalog, France
Robin Messer, Altran-Praxis, UK
Roger Brandt, Telia, Sweden
Claus Stellwag, Elektrobit AG, Germany
Quentin Ochem, Ada Core, France
Martyn Pike, Ada UK

Call for Regular Papers

Authors of regular papers which are to undergo peer review for acceptance are invited to submit original contributions. Paper submissions shall not exceed 14 LNCS-style pages in length. Authors shall submit their work via EasyChair following the relevant link on the conference web site. The format for submission is solely PDF.

Proceedings

The conference proceedings will be published in the Lecture Notes in Computer Science (LNCS) series by Springer, and will be available at the start of the conference. The authors of accepted regular papers shall prepare camera-ready submissions in full conformance with the LNCS style, not exceeding 14 pages and strictly by March 29, 2015. For format and style guidelines authors should refer to http://www.springer.de/comp/lncs/authors.html. Failure to comply and to register for the conference by that date will prevent the paper from appearing in the proceedings.

The CiteSeerX Venue Impact Factor has the Conference in the top quarter. Microsoft Academic Search has it in the top third for conferences on programming languages by number of citations in the last 10 years. The conference is listed in DBLP, SCOPUS and Web of Science Conference Proceedings Citation index, among others.

Awards

Ada-Europe will offer honorary awards for the best regular paper and the best presentation.

Call for Industrial Presentations

The conference seeks industrial presentations which deliver value and insight but may not fit the selection process for regular papers. Authors are invited to submit a presentation outline of exactly 1 page in length by January 25, 2015. Submissions shall be made via EasyChair following the relevant link on the conference web site. The Industrial Committee will review the submissions and make the selection. The authors of selected presentations shall prepare a final short abstract and submit it by April 12, 2015, aiming at a 20-minute talk. The authors of accepted presentations will be invited to submit corresponding articles for publication in the *Ada User Journal (http://www.ada-europe.org/auj/)*, which will host the proceedings of the Industrial Program of the Conference. For any further information please contact the Industrial Chair directly.

Call for Tutorials

Tutorials should address subjects that fall within the scope of the conference and may be proposed as either half- or full-day events. Proposals should include a title, an abstract, a description of the topic, a detailed outline of the presentation, a description of the presenter's lecturing expertise in general and with the proposed topic in particular, the proposed duration (half day or full day), the intended level of the tutorial (introductory, intermediate, or advanced), the recommended audience experience and background, and a statement of the reasons for attending. Proposals should be submitted by e-mail to the Tutorial Chair. The authors of accepted full-day tutorials will receive a complimentary conference registration as well as a fee for every paying participant in excess of 5; for half-day tutorials, these benefits will be accordingly halved. The *Ada User Journal* will offer space for the publication of summaries of the accepted tutorials.

Call for Workshops

Workshops on themes that fall within the conference scope may be proposed. Proposals may be submitted for half- or full-day events, to be scheduled at either end of the conference week. Workshop proposals should be submitted to the Conference Chair. The workshop organizer shall also commit to preparing proceedings for timely publication in the *Ada User Journal*.

Call for Exhibitors

The commercial exhibition will span the three days of the main conference. Vendors and providers of software products and services should contact the Exhibition Chair for information and for allowing suitable planning of the exhibition space and time.

Grants for Reduced Student Fees

A limited number of sponsored grants for reduced fees is expected to be available for students who would like to attend the conference or tutorials. Contact the Conference Chair for details.

ACM DL DIGITAL LIBRARY

Add the ACM Digital Library to your membership—or join ACM and get the DL at member rate

The ACM Digital Library is simply the world's largest, most respected online resource for computing professionals. The DL includes the full text of more than 88 ACM publications, including journal papers and magazine and SIG newsletter articles, plus proceedings from more than 500 annual conferences…all at your fingertips, from desktop or mobile platforms.

With a DL subscription, you can enjoy unlimited access to an integrated bibliography covering the entire computing field:

- Stay on top of the latest innovations from top thought leaders, researchers, entrepreneurs, and makers in cutting-edge technological fields
- Discover new ideas they shared at more than 170 SIG-sponsored events around the world—scan a paper, or watch a keynote talk
- Search across a comprehensive computing bibliography of more than 2.3 million records from over 5,000 publishers worldwide

Available to ACM Members only

Current ACM Professional Members
can add the ACM Digital Library for only $99

Or join ACM now and include the DL at the member rate—$198
Join ACM online at www.acm.org/joinacm

To subscribe to the ACM Digital Library, contact ACM Member Services:

Phone: 1.800.342.6626 (U.S. and Canada)
+1.212.626.0500 (Global)
Fax: +1.212.944.1318
Hours: 8:30 a.m.-4:30 p.m., Eastern Time
Email: acmhelp@acm.org
Mail: ACM Member Services
General Post Office
PO Box 30777
New York, NY 10087-0777 USA