RAPID PROTOTYPING IN ADA IN THE RATIONAL ENVIRONMENT EMPHASIZING SOFTWARE REUSE

Peter H. Luckey and Frank G. DuPont

IBM/FSD, Route 17C, Owego, NY 13827

INTRODUCTION

A recent experience at IBM/FSD Owego demonstrates how prototyping in Ada is enhanced via the incorporation of software reuse technologies in an integrated development environment. In response to a recent new business proposal at Owego, a user-interface for a data-base application was prototyped. The purpose of the prototyping exercise was three fold;

1. To aid in the size estimation of a program to be developed
2. To confirm the viability of developing the program in Ada
3. To demonstrate the productivity possible when developing with reuse in mind in the Rational Environment.

The results of the exercise were that the purpose was accomplished and an object-oriented prototyping process was developed.

Two software engineers with knowledge of both reuse techniques and available reuse libraries, working in the Rational Environment, were able to complete a 6 thousand source lines of code (KSLOC) prototype in 27 man-hours. The results, summarized in Table 1-1, indicate that 70% of the prototype's SLOC consisted of reusable components integrated, without alteration, from external libraries. Of the remaining 1.8 KSLOC, an estimated 50% was developed using code templates. This consisted of developing and testing a template then altering localized portions of the template for each instance of it.

Table 1-1. Prototype SLOC Summary

Code Classification	SLOC	Hours
Reused	4,209	1
Prototype (50% via templates)	1,797	26
Total	6,006	27

ENVIRONMENT

The software development environment was one of the keys to the success of this prototype effort. The development environment is comprised of an integrated collection of hardware and software. Figure 1-1 illustrates the hardware configuration.

Hardware

Each software developer has an intelligent workstation on his desk, either an IBM PC/AT or PS/2. Each of these workstations is a node on an IBM Token Ring local area network. Control units are also connected to the Token Ring so that the workstations may be used as terminals for the IBM mainframes. An IBM PC/RT is used as a gateway connecting the Token ring with an Ethernet. Also on the Ethernet are two Rational R1000, Series 200, Model 20s and an IBM 8232. Each of the workstations may be used as a terminal for one of the R1000s. The IBM 8232 provides a high speed connection between the R1000s and one of the IBM mainframes. Not pictured are the various printers, tape drives, etc. which are also part of the development environment.

Software

The software available on each of the processors in the environment and the software used in connecting the processors is integral to the success of the development environment. The largest part of the development is performed in the Rational environment. The R1000s play host to a truly integrated set of software development tools for Ada including: a syntax and semantics directed editor, a document generator, a consistency and completeness checker, a program cross reference and traversal tool, configuration and version control tools, a debugger, a compiler, VM/370 and DOS/PC cross development tools, and many others. The Ada tools available in both the VM/370 and DOS/PC environments are provided by Alsys. The prototype's target Ada compiler was the Alsys PC AT Ada compiler Version 3.2.

Reuse Resources

There are three reusable Ada software libraries in the development environment; an IBM corporate library, an Owego site library, and a specific project library.

1. Reusable Component System (RCS)

   RCS was developed at IBM/FSD Houston. This library system for reusable components is an extensive prototype for the Corporate Reuse Environment (CRE) which is currently under development. RCS groups components into classes according to a domain analysis. This classification eases the search and retrieval process. The library currently

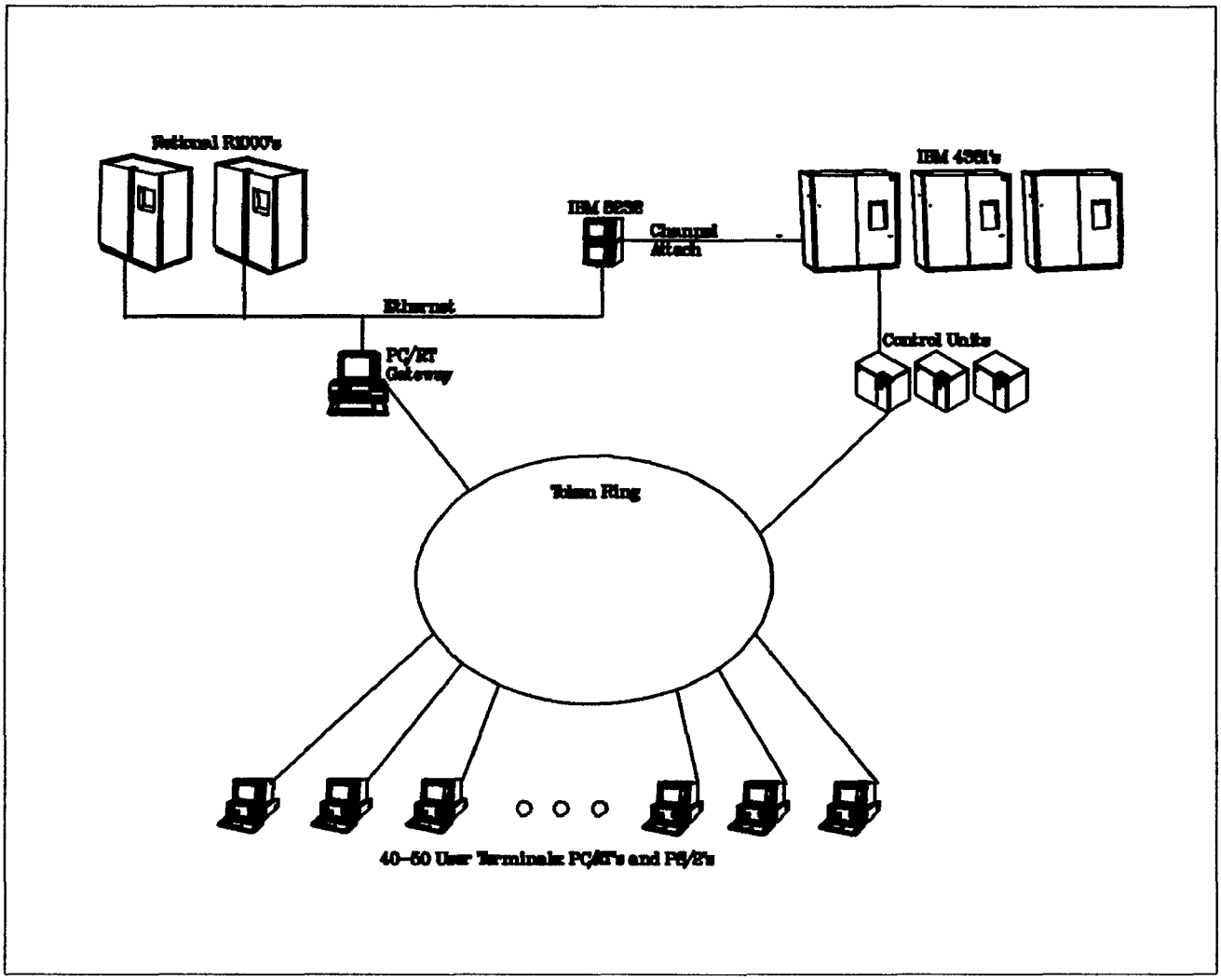

FIGURE 1-1. Software Development Environment

contains the Booch and Karlsrue generic abstract data type and utility components.

2. Owego Reusable Software Library (RSL)

   The Owego RSL is an out-growth of the Owego Software Reuse Quality Improvement Team (QIT) efforts. The establishment of the RSL precedes that of RCS. Included in the RSL are design support components, generic abstract data type and utility components, and components derived from a reusable avionics project.

3. Reusable Components from a recent Owego project

   A recent Owego project's development plan included a strategy for incorporating software reuse into the design and coding phases of the development life cycle. As part of the strategy a project specific RSL was created. The reusable components in this library include generally reusable components as well as components that are designed to be reused solely within the project. The reusable components in the project RSL will eventually be considered for inclusion in either the corporate or site RSL.

PROCESS

The design process that is typically followed for high level design is shown in the following PDL/Ada procedure. The design process is object-oriented, and is based on Grady Booch's Object-Oriented Development Method[1]. Once the high level design is complete, detailed design is performed via a stepwise refinement of the object interface specifications, taking into account target limitations. Coding is simply a stepwise refinement of the detailed design. Extensive unit, module, and integration testing is performed bringing t he development process to its conclusion. The whole process is an iterative process, and can easily be accommodated by either the traditional waterfall model or the spiral model for the software development lifecycle.

[1] "Software Engineering with Ada", Second Edition, Chapter 5.

```
procedure Object_Oriented_Design (From_The : Problem_Definition) is
begin
  Process:
    loop
      loop
        -- Develop an informal strategy from the
        -- problem definition.
        Develop (Informal_Strategy, From_The);

        -- Review the informal strategy.
        Review (Informal_Strategy, Returning => The_Outcome);
        exit when The_Outcome = Passed;
      end loop;

      -- Formalize the strategy by going through the
      -- following eight steps.

      -- 1) Identify the objects and their attributes
      Identify (List_Of_Objects, From_The => Informal_Strategy);

      -- Exit the process loop when no new objects are identified.
      exit Process when Is_Empty (List_Of_Objects);

      -- 2) Identify the operations.
      Identify (List_Of_Operations, From_The => Informal_Strategy);

      -- 3) Associate operations with objects.
      Associate (List_Of_Operations, With_The => List_Of_Objects);

      -- 4) Search for a reusable component to take
      -- the place of each newly identified object.
      for Each_New_Object in List_Of_Objects loop
        -- Search for a reusable component.
        Search (Reusable_Component, Like => Each_New_Object,
                Returning => Reusable_Component_Found);
        if Reusable_Component_Found then
          -- Remove the object from the object list.
          Remove (Each_New_Object, From_The => List_Of_Objects);
          -- Add the reusable component to the design.
          Add (To_The_Design => Reusable_Component);
        end if;
      end loop;

      -- 5) Establish the visibility of each object.
      Establish_Visibility (For_The => List_Of_Objects);

      -- 6) Establish the interface of each object.
      Establish_Interfaces (For_The => List_Of_Objects);

      -- 7) Record the formalization of the strategy in
      -- graphical representations and PDL/Ada.
      Generate_Graphics (From_The => List_Of_Objects);
      Generate_Pdl_Ada (From_The => List_Of_Objects);

      -- 8) Iterate through the OOD process for each
      -- newly identified object.
      for Each_New_Object in List_Of_Objects loop
        Object_Oriented_Design (Each_New_Object);
      end loop;
    end loop Process;
end Object_Oriented_Design;
```

Obviously, following the detail of the steps outlined above would defeat the major purpose of rapid prototyping. It is more important to readily produce a model that proves or disproves the viability of a concept. Therefore, a generalized method for developing a rapid prototype was followed. This method, described below, is based on the more stringent development process, but eliminates or modifies some of the more time consuming elements.

1. Define the problem to be solved by the prototype.

   When not working on a prototype the inputs to the problem definition are typically a formal set of requirements and specifications. When working on a prototype these sources are most likely informal. The problem definition outlines in concise terms the problem that is to be solved.

2. Identify the major objects in the system.

   Notice, there is no informal strategy recorded and reviewed. The informal strategy is most likely in the mind of the prototype designer only recorded via scratchy notes. The object identification becomes natural to the software engineer experienced with object-oriented methods.

3. Identify reusable components which might perform the work of the major objects.

   There may not be an exact fit, but implementing a prototype typically means there is a time constraint. There is not time to develop from scratch and little time can be devoted to changing a component to do exactly what is needed. Compromising needs to take place. It is most important that the developing engineers are aware of reuse resources available and to determine a way to use them, even if there is not an exact fit.

   Not only should "pure reuse", reuse without altering code, be considered; but also, other forms of reuse which might involve altering code slightly. The other forms of reuse may need to be utilized because there is insufficient time to do pure reuse. For example, there is not time to develop an Ada generic package that meets all of the requirements. The other forms could not only save time but should give good information about how to plan for "pure reuse" when implementing the real system.

4. Establish the visibility and the interfaces among the objects.

   High level design is concerned with establishing the structure or architecture of the system and determining the interfaces between the structural components.

   - If a reusable component was identified to act as one of the major objects, determine which services of the component will be needed. Become familiar with the interface to these services so that they may be used effectively.
   - If a major object must be developed, identify the operations to be performed by the object. Determine how this object will make use of the reusable components. Iterate through this abbreviated Object-Oriented Design process with the object in question.

5. Perform detailed design of the prototype

   The detailed design is performed via a stepwise refinement of the "to be developed" object interface specifications. It turns out that two elements in the prototype are being refined; the major control flows for the system, and the "glue" to hold all the structural components together. During the stepwise refinement process, similar patterns in the logic might become evident. The patterns indicate that a generic or reusable component could be developed. If time does not exist to develop the component for the prototype, at least a template should be created so there is some form of reuse.

   During detailed design smaller reusable components should be identified to perform common functions. These types of components are often utility packages (i.e., String_Utilities, Time_Utilities, Integer_Utilities, or even Text_Io). Remember to look for places where these types of components can be used instead of re-implementing the function despite its simplicity.

6. Code via stepwise refinement of the detailed design.

7. Informally test, mainly at the integration level.

   The majority of a prototype system should be made of reusable components and the "glue" to hold them together. The reusable components should not need unit or module testing. Therefore, the testing should stress the "glue", and this is most effectively done during integration test. Using Ada, many of the discrepancies that would come out during unit and module test are detected by the compiler.

EXAMPLE

The following is a description of the actual process applied to the prototype. Both engineers working on the prototype were familiar with object-oriented development methods, the available software reuse resources, and the software development environment.

The problem definition for the prototype was based on two inputs. The first was simply the idea to develop a prototype to illustrate the feasibility of converting the program to Ada. The second input was the existing program. The problem definition was to replicate the top level screen displays and the screen traversal mechanism in Ada.

Based on the problem definition, two major objects were identified.

1. Full-screen display manager
2. Screen traversal mechanism.

Two reusable components were identified which might perform the work of the major objects. Display_Manager was found in the Owego RSL to act as the full-screen display manager. To handle the traversal of a hierarchic menu structure, a Tree_Abstract_Data_Structure was retrieved.

A user's guide was available for Display_Manager, so that and the package specifications were used to become familiar with the capabilities of and interface to the reusable component. This raised a concern regarding the time necessary to update screens. The concern was that the definition of a single field on a screen would require between three and ten Ada procedure calls. The code would tend to be error prone: fields at incorrect locations on the screen, or the wrong size, or overlapping one another, or misspelled words etc. Fixing these types of problems would require altering, recompiling, re-linking and re-executing the Ada code. Even with a small number of screens, this could account for a fair amount of Ada code. Therefore, in order to implement the screens in a timely manner, a tool was defined which would create screens based on instructions in an external, flat file. This would allow alterations to a screen to be done by editing a flat file, and rerunning the tool.

Because Display_Manager is a PC based component and the major part of the development was to take place in the Rational environment, Display_Manager had to be ported to Rational. Display_Manager is also PC dependant so those dependencies had to be stubbed out to allow for design, development and test on Rational.

With a development environment set up, design of the tool component began. The component has two major functions.

1. Read a flat file
2. Create screen (based on instructions from flat file)

Breaking the second function down further produces the results below.

1. Read a flat file
2. Create screen (based on instructions from flat file)
   a) Parse a line
   b) Call Display_Manager (based on instruction in line)

This break down indicated there were two other components needed: a line parser and an external file manager. The two components were retrieved from the Owego RSL and the project RSL respectively.

With the three necessary reusable components available, the tool's detailed design began by gluing the components together. During this phase, the need for another small component was identified. The tool was required to prompt the user for two pieces of information. There is a small reusable component, Get_String, which displays a message (passed as a parameter), prompts the user for input and returns the user's response. This component was retrieved and incorporated into the tool. Tools to display a screen and delete a screen were quickly written. They also incorporated the Get_String function.

With tools implemented and tested, high level design of the first major object, full-screen display manager, was complete. High level design of the second major object, the screen traversal mechanism, started with a familiarization process with the Tree_Abstract_Data_Structure component specification. It was determined that the traversal mechanism could be more easily implemented using the built-in call stack.

At this point development proceeded in two parallel paths. An external file for each of the sixteen screen displays had to be created. The files contained the field definitions for the screens. The second path entailed performing the detailed design. The detailed design and coding of the prototype began by coding the control structures for the main menu and stubbing the submenus. This was tested on Rational and ported to the PC AT. No problems were discovered, so the approach seemed feasible.

Implementing the low level packages (submenus) was then started. The cycle was the same as with the tool and top level of the prototype. Design, implementation, and unit and some string tests were performed in the Rational environment. Final integration test was performed on the AT. Bugs found at integration test were corrected on Rational. Only the files containing the affected code were recopied to the AT and recompiled. This process is automated within the Rational Environment. The design of the submenu packages showed a great deal of similarity between the main menu package and other submenu packages. This lead to the conclusion that control structures for a screen display could be encapsulated in an Ada generic package and instantiated for each display. This solution did not appear to be time effective. Therefore, instead of creating the generic and using "pure reuse" (reuse without code alterations), another form of reuse was adopted. This consisted of copying the tested implementation of one submenu into a new submenu under development and then altering the code. Types of structures that had to be changed include: constant values, alternatives of case statements, subprogram calls, etc.

RESULTS

The amount of code developed for the prototype and the amount of time required to design, implement and test are listed in the table below.

Developed Code	SLOC	Hours
Tools	211	6
Prototype (50% via templates)	1,586	20
Total	1,797	26

The components that were reused and their associated SLOC count are listed in the table below. Not every available function in each component was used in the prototype (for instance String_Utilities.Make_Lowercase was never called), but the SLOC count represents the entire amount of code for each component.

Reused Components	SLOC
Display_Management	
--------Ada	2,628
--------PC Assembler	1,752
External_File_Management	765
Line_Parser	110
Get_String	10
String_Utilities	696
Total	
--------Ada	4,209
--------PC Assembler	1,752

Sixteen screens were implemented, constituting approximately 270 fields of data. The table below illustrates that screen definitions require more lines in flat files than it is estimated they would have required in Ada SLOC. Even with this fact, it is believed the tools saved time. Using the tool, changes to screen layout required editing a flat file and rerunning the tool; while not using the tool would have required altering Ada source, recompiling and re-executing the program.

Menu Definition Files	
External Flat Files	3,509 lines
Ada code estimate	3,000 SLOC

CONCLUSION

There are two categories of conclusions that can be drawn from the prototype experience. First are the conclusions specific to the prototype developed. Second are more general conclusions regarding rapid prototyping in Ada in the Rational Environment where there is an emphasis on software reuse.

Prototype Specific

The primary objective for developing the prototype was to substantiate the size estimates for implementing the language and database conversion to Ada with a prescribed relational database. Originally, the plan for the prototype included interfacing with the database via an Ada SQL program interface. This was not possible because the interface was not available. Therefore, the prototype provided valuable data regarding the size of the effort to convert the multiple display screens of the system and the mechanism for traversing the screen hierarchy.

Secondary objectives of developing the prototype were to confirm the viability of converting the system to Ada, and to illustrate the possible development productivity when developing in the Rational environment with established reusable Ada software libraries available. The fact the prototype was completed confirms the viability of converting the user interface and control structures to Ada. The numbers presented can be used to calculate a productivity rate, in the order, of 2000[2] SLOC per man-month. The calculation takes into account the assumption that the coding phase is less than 20% of the total development lifecycle. This indicates that experienced software engineers developing an Ada system in an integrated software development environment where software reuse is emphasized can reasonably achieve productivity rates, in the order, of 500 to 1000 SLOC per man-month.

General

Even though this was a relativly small exercise, and realizing that more experimentation should be performed with larger prototypes, a number of general conclusions may be drawn.

- The design methodology adopted, when prototyping or not, should incorporate a software reuse strategy. The prototype is comprised of approximately 6000 Ada SLOC, of which only 1500 SLOC was newly developed. All of the functions available with the reusable components were not used in the prototype. Therefore, less than 30% of Ada SLOC in the prototype was newly written, yielding a 70% pure reuse factor. Of the newly written SLOC, approximately 50% was developed using a less pure form of reuse.
- Especially if the implementation language is Ada, and if the effects of software reuse are to be maximized, an object-oriented development method should be used. An object-oriented development method facilitates reuse. The greatest gains in productivity on this project were made by reusing objects, available in an RSL, identified during the object-oriented design process.
- A large part of the productivity gains are directly attributable to the use of the integrated, Ada and object-oriented, tool set provided by the Rational Environment.
- The final conclusion is that these methods, tools, and reuse resources must be understood and incorporated by trained and experienced software engineers.

---

[2] The calculation was based only on the amount of code developed and included in the prototype. 20 hours is approximately 1/8th of a man-month. The prototype activity covered only 1/5th of the development lifecycle activities. (1,586 SLOC * 8) / 5 = 2537 SLOC per man-month.