Mark Stephen Bond

Library Packages for Ada 95

CS380−IP

Final Year Individual Project

Project Report

B.Sc. Computing Science

May 2000

School of Engineering and Applied Science

Aston University

Birmingham

Supervised by Dr. Alan Barnes

B.Sc. Computing Science
Final Year Project Definition

Student's name Mark Bond
Project Title AB5 − Library Packages for Ada 95
Principle objectives Extend the generic Ada 95 abstract data types provided by the Booch Components using the advanced object−oriented features of the Ada 95 programming language. Concentrate initially on persistent versions of existing data structures and, if time allows, consider implementing new data−structures (e.g. dequeues and priority−queues).
Environment for the work (e.g. computer, language to be used) **Language**: Ada 95. **Compiler**: GNU Ada Translator (GNAT). **Operating system**: POSIX compliant − Linux (home) and Unix (university). **Machines**: PC (home) and Sparc (university).
Timetable showing main stages in work plan **October 1999**: Preliminary research and analysis. **November 1999**: Draw up specification and start design / implementation. **December 1999**: Provide written Progress Report . Continue implementation. **January 2000**: Continue implementation, testing and documentation. **February 2000**: Review progress to finalise scope of project. **March 2000**: Continue implementation, testing and documentation. **April 2000**: Completion of code and Final Report . **May 2000**: Printing and binding the project. Final preparation for the presentation.

Student's signature ______________________________ *Date* ____________________

Supervisor's signature ___________________________ *Date* ____________________

Acknowledgements

Thanks are due to Grady Booch (creator of the original C++ components), David Weller (who ported the C++ components to Ada 95 and Simon Wright (current maintainer of the Ada 95 components) without whom this project could never have transpired.

Thanks also go to Dr. Alan Barnes, who supervised this project and whom, like Simon Wright and Robert Leif, offered helpful advice during the course of the project.

Finally, thanks go to Hamish Miller for the time he spent proof reading this report and the suggestions he gave.

Table of Contents

Table of Figures

1 Structure of the report

The report is organised as a number of self contained chapters covering the following topics:

- **Introduction** − An introduction of basic concepts used throughout the report and a brief statement of the aims of the project.
- **Context** − A description of the motivation for and the background of the project along with a discussion of the work performed in order to determine the project requirements.
- **Requirements** − A statement of the requirements that were to be achieved by the project.
- **Design** − A description of the general structure and features of the system that was to be implemented.
- **Implementation** − A high−level description of the system implemented and low−level details of noteworthy aspects along with testing methods and their outcomes.
- **Evaluation** − An explanation of the evaluation methods applied to the end−product and a discussion of the success of the project in terms of how well the original objectives were realised and the reliability and usability of the end−product.
- **Conclusion** − A brief reiteration of the work done including a description of the problems encountered and acknowledgement of limitations of the project along with recommendations as to how the project could be taken further in order to eliminate any such limitations.
- **References** − A list of works referenced throughout the report.
- **Bibliography** − A list of works consulted but not explicitly referenced in the report.
- **Appendices** −User instructions (how to install and use the components). A list of source files. Examples of code written. Outcome of test programs. Communications made during the course of the project.

2 Introduction

An introduction of basic concepts used throughout the report and a brief statement of the aims of the project.

2.1 The Booch Components

Grady Booch, currently an employee of the Rational Software Corporation (Methodologists−−Rational University: Education & Training, Software Development 2000), is a leading figure in object−oriented software methodologies. He jointly developed the Unified Modeling Language (UML) − widely adopted as the industry standard language for the specification, implementation and documentation of the features of software systems − and was responsible for the Booch Method, a leading object−oriented analysis and design methodology. His work has made significant contributions to improving the effectiveness of software developers world−wide.

The Booch Components (Wright 1999), a set of libraries implementing a number of highly generic abstract data structures, were also a remarkable development attributed to Grady Booch. These object−oriented ADTs were key to demonstrating the feasibility of code reuse, helping to make it a key aim of software engineering today. Coded in C++, the Booch Components are made up of three key abstractions:

1. **Tools** − providing algorithmic abstractions such as sorting and searching.
2. **Support** − providing the concrete structures from which the components are implemented.
3. **Structs** − providing abstractions of the containers implemented by the Components.

The Components implemented an extensive range of containers including bags, collections, dequeues, graphs, lists, maps, queues, rings, sets, stacks and trees and hence became a valuable asset of many C++ developers.

In 1994−1997 David Weller (dweller@riva.com) created an Ada 95 port of the Booch Components. Other members of the Ada community contributed to the Ada 95 Components including Pat Rogers (email; pat@classwide.com), of Software Arts & Sciences (Classwide 2000), who added storage management functionality in 1998, and Simon Wright (Wright 1999) who now officially maintains the Ada 95 Components after taking over from David Weller in 1997.

As a consequence of the architectural differences between Ada 95 and C++, various differences developed between the two variants of the Components − each variant taking advantage of the particular strengths of the relevant language. Features present in Ada 95 that were not in C++ (such as safe generics and object−oriented programming, access discriminants and concurrency) as well as those features present in both languages (e.g. user−defined storage management, garbage collection and inheritance) led to the development of highly advanced and flexible containers.

As an example of the internal complexity of the containers (necessary to heighten the abstractness and ease of use presented to the client) consider the following features of the Ada 95 Components:

- **Iteration** − Every container implemented in the Ada 95 Booch Components is supported by routines allowing both active and passive iteration. Active iteration allows user controlled traversal of containers using commands such as "Next" to move from one data element to the next. Passive iteration takes the abstraction even further by allowing a client defined routine to be applied to all the items of a container in a single command.
- **Concurrency** − Some containers of the Components are implemented in guarded and synchronised forms to control access in concurrent systems. Guarded forms provide a semaphore to protect each instance of the container whereas synchronised forms effectively perform each access to a container as an atomic transaction.

2.2 The project

The aim of this project was to extend the Ada 95 Booch Components in order to support containers exhibiting persistence. Persistence was the one identified feature of the original C++ Booch Components that was not ported across to the Ada 95 variant.

Two types of container exhibiting persistence were identified with different characteristics. For clarity, consistent use of two distinct names ("persistent container" and "external container") have been used throughout this report in order to identify the two types of container. The different characteristics of these two container types are described below:

- A "persistent container" was defined as a data structure using main memory for its primary storage. All the data elements of a persistent container are held in memory during program execution. A persistent disk file associated with the container is referenced only twice during the active lifetime of the object − once as data is loaded from the file into memory and once as data is stored from memory into the file.

- An "external container" was defined as a data structure primarily using external storage (an associated disk file). Only the data elements currently being accessed by the program are loaded into main memory.

In addition to "persistent containers" and "external containers" (jointly referred to as "non−volatile containers"), the report makes references to those containers that already form the existing Booch Components and hence exhibit no persistent characteristics. These containers are consistently referred to as "volatile containers".

3 Context

A description of the motivation for and the background of the project along with a discussion of the work performed in order to determine the project requirements.

Motivation for this project began with a keen interest in object−oriented programming acquired from previous exposure to the Turbo Vision libraries of Turbo Pascal. Using these libraries made me aware of the power offered by object−oriented techniques with particular respect to abstraction and code re−use.

Further motivation grew from admiration of the Ada 95 programming language, in particular the manner in which it encourages the development of code that is both reliable and readable without restricting what can be easily achieved.

The original release of the Ada programming language (now known as Ada 83) in 1983 was not object−oriented but did offer support for object−based programming. Developers could use packages to create ADTs by providing access routines to view and modify the content of the data structure and enforcing no direct access to the underlying implementation of the structure. As a programming strategy this was highly successful in terms of increasing code understandability but was only a step towards what could be achieved with a fully object−oriented approach.

In 1995 object−oriented features were added to Ada with the second release of the language (Ada 95). This release incorporated inheritance and polymorphism providing developers with a wide range of benefits. Such features allowed for substantial reductions in code duplication, increases in code reuse and simplified extension of existing code.

Other features of Ada making it an attractive tool for development include its advanced exception handling mechanism and its strict strong typing rules which considerably reduce the risks of program development compared with less rigid languages such as C and C++.

A combination of the above motivations led to an initial decision being made − that the project would be an investigation into the advanced object−oriented features of the Ada 95 programming language. Further investigation was required to determine the best project topic that would enable this aim to be achieved.

Initial investigation began with email communications to potential project supervisors before the deadline for deciding upon a project. One such communication, from Dr. Alan Barnes, mentioned the Ada 95 Booch Components and prompted further investigation. Subsequently, the current maintainer of the Booch Components (Simon Wright) was contacted. Simon’s reply (see "Simon Wright", page 84) confirmed my suspicion that a useful feature of these components was missing − i.e. the option of non−volatility. The opportunity was thus taken to both investigate the object−oriented features of Ada 95 whilst, at the same time, performing the worthwhile task of coding non−volatile versions of the data−structures in the Booch Components. This decision was made early on during the project and effectively introduced a change in nature of the project from the original specification[1]. The original intention had been to code packages from scratch but the discovery of the Ada 95 Booch Components led to the realisation that it would be potentially more beneficial to expand upon existing packages.

As discussed in the "Introduction" (page 2) two different types of non−volatile container were identified. A final phase of preliminary investigation was therefore required in order to decide which type of non−volatile container to produce. The result of this investigation was that a decision was made to concentrate primarily on coding persistent containers assigning a lower priority to coding external containers. Justification of this decision follows:

- Persistent containers clearly offer the potential for a huge a speed advantage over external containers since the data elements are held in fast main memory rather than slower external memory.
- Persistent containers can be closely based on the existing volatile containers. This could allow existing code of the volatile containers to be reused thus ensuring a common interface between volatile and non−volatile containers as well as reducing coding and testing effort.
- The potential disadvantage of persistent containers (i.e. being more limited in size than external containers due to the limited amount of main memory) was considered less important than the advantages listed above. This was deemed so since most modern operating systems implement virtual memory which provides an effective main memory size limited only to the volume of external storage. Furthermore, the identified missing feature of the Booch Components was a lack of non−volatile containers rather than a need to increase the maximum capacity of the existing containers.

1 This change in project nature was supported by my project supervisor.

- Not all operating systems implement a virtual memory system and hence, in some situations, external containers could be preferable to persistent containers. For this reason it was decided to implement these as a secondary objective.

Given Ada 95's strongly defined mechanisms for interfacing with library packages and its popularity in academic environments, it was not surprising to discover an extensive selection of freely and commercially available library packages supporting Ada 95 development. One such existing work was brought to my attention as a consequence of its close relation to the project. The library, named PragmARC − PragmAda Reusable Components (Carter 2000), provided various utilities for Ada programmers including some abstract data types as well as additional features such an ANSI TTY control. Some of the ADTs implemented by the PragmARC libraries (including queues and lists) overlapped with those of the Booch Components but PragmARC was found to offer no support for persistence. Given PragmARC's generally less sophisticated implementation of ADTs (which made no use of tagged types) extension of these packages would have been more problematic that that of the Booch Components.

4 Requirements

A statement of the requirements that were to be achieved by the project.

The project description, as given in the document listing the options available for Final Year Project topics, was as follows:

> The project involves developing a number of packages to support operations with dynamic data structures and / or advanced sorting and searching methods. This project would involve a literature search on advanced algorithms (for example on searching, sorting, tree balancing etc.) and then implementing Ada versions of these algorithms. The project would involve the student in extending their knowledge of topics covered in CS121 (Data Structures and Algorithms): ADTs, generics, recursion etc. Although it would not be essential an ambitious student might also investigate and utilise the object−oriented features in Ada in the project work. Practical work using the GNAT Ada compiler on University Unix systems would be involved.

The above description, however, was intended only as a general guide as to what the project would involve and was not intended as a formal requirements specification. More specific requirements were to be determined after performing some background investigation (see "Context", page 5).

As a consequence of the missing support for non−volatile containers in the existing Ada 95 Booch Components and my interest in the object−oriented feature of Ada 95, the following primary objective was drawn up for the project:

Extend the abstract data structures implemented within the Ada 95 Booch components in order to provide support for persistent containers. Take advantage of the advanced object−oriented features of the Ada 95 programming language wherever necessary in order to maximise the generic flexibility of the persistent containers. Object−oriented techniques should also be used to ensure that the code is efficient (in terms of re−using the existing tested code) and offers extensibility should new needs arise for the Components in the future.

A secondary objective was drawn up for implementing external containers. As discussed in "Context" (page 5), the benefits offered by external containers are less significant than those of persistent containers but it was decided that, if time permitted, the coding of a small number of external containers would prove beneficial. Not only could these external containers prove useful in their own right but they would also provide a framework or template for the coding of the additional external containers should anyone else wish to continue this work.

5 Design

A description of the general structure and features of the system that was to be implemented.

Since the project was to extend an existing set of Ada 95 packages, the design of the modifications would be heavily influenced by the design of the existing code. It was therefore necessary to ensure a complete understanding of the existing packages so as to ensure that the additions would not introduce any unnecessary inconsistencies or limitations.

It was decided that no changes would be made to existing files of the Booch Components. All the non−volatile containers were to be implemented by coding additional files rather than by editing current files. This was decided so as to eliminate any possibility of changes that could adversely affect the operation of any existing programs making use of the volatile containers. This requirement further amplified the need for a thorough understanding of the existing components so as to gain knowledge of how the existing packages could be extended without editing existing files − for example through the introduction of child packages (Barnes 1996, p. 238).

5.1 Structure

In general, the overall structure of the existing volatile containers of the Booch Components was found to be as shown in "Figure 1" (page 11). This BON diagram represents a selection of containers (queues, stacks and sets) of the Components that represent its overall structure. With a few exceptions, the volatile components are derived from a single abstract tagged object ("Container"). For each container type ("Stack", "Queue", "Set" etc.) there may be several concrete implementations (usually "Bounded", "Dynamic" and "Unbounded"). Support packages exist that provide routines specific for each of the implementation types (e.g. "BC.Support.Bounded") and for additional common support routines such as hash table support and storage management.

It was decided to try and maintain a similar structure when implementing the new non−volatile containers and to take advantage of any possible benefit offered by the existing design. The common ancestor ("Container"), for example, from which most containers are derived proved very useful by allowing a generic "Persistent_Container" tagged type to be created.

Figure 1 − BON diagram of existing components.

5.2 Data items

The generic parameter "Item" that defines the data items for containers of the Booch Components is declared as:

```
    TYPE Item IS PRIVATE;
```

It is therefore ensured that "Item" is definite (i.e. uninitialised objects of type "Item" can be declared) and that equality and assignment are defined (Barnes 1996, p. 370).

The definition of "Item" therefore restricts the allowed data items for the existing volatile containers, preventing them from working with limited types or indefinite types such as abstract types, unconstrained arrays, variant records or discriminant records.

Given this existing limitation of the Booch Components, it was decided that the implementation of the non−volatile containers should ideally make no further restrictions on the allowed data items but would also not be required to relax this restriction (for example by supporting limited types). Further details of data items supported by the implemented non−volatile containers is given in the "Implementation" section (see "Data items" page 18).

5.3 Persistent containers

5.3.1 Overview

Persistent containers were to be implemented as data structures using main memory for primary storage. They were to have persistent disk files referenced only at the start of the objects' active lifetimes (when all data elements were to be loaded from file to memory) and at the end of the objects' active lifetimes (when all data elements were to be stored back to file).

The behaviour and implementation of a persistent container was therefore to be identical to that of the equivalent volatile variant throughout most of the program execution − the only difference being the additional functionality to perform the initial data load and the final data save. In order to best achieve this, it was therefore decided that each persistent container should be a tagged type derived from the corresponding volatile container.

It was also preferable that code duplication was kept to the minimum. In order to help achieve this, it was decided that all functionality common to all persistent containers would be extracted out into a shared package. This led to the development of the abstract tagged type "Persistent_Container" from which all persistent containers were derived.

5.3.2 Persistent data

The obvious solution decided upon for storing the data items in a persistent manner was to make use of disk files. One file was to be associated with each persistent container from which the data would be read when the object was initialised and to which it will be written back when the object was finalised. This was not the only considered solution (see "Memory mapping", page 26 for an alternative) but all deliberated solutions did involve the use of some sort of disk file to store the persistent data. Several options were therefore considered relating to methods of specifying the name of the associated file as described below:

1. The filename to be used could be supplied as a generic parameter to package, e.g.

   ```
   GENERIC
      File_Name : STRING
   PACKAGE BC.Containers.Stacks.Bounded.Persistent IS
      ...
   END BC.Containers.Stacks.Bounded.Persistent;
   ```

   This method was discarded because it would have required separate instantiations of the generic package for each instance of a container. This would have thus introduced various complexities such as preventing comparisons between containers.
2. The filename could have been automatically determined from the name of the variable holding the container. This method was discarded because of the following problems:
   - The rules for valid variable names do not match the rules for valid filenames.
   - The scope of variables does not match the scope of files. Hence, two separate variables may have had the same name but required two distinct files.
   - No method of determining the name of a variable in Ada could be determined.
3. The filename could be provided by issuing a procedure call to the ADT. This method was discarded because of the following problems:
   - There is no correlation between variable declarations and procedure calls. Hence containers could have been assigned filenames several times or not at all and name assignment may not have always been performed before the container was first used.
   - Copying one container to another (using ":=" assignment) would have resulted in the filename associated with the target container being lost.
4. The filename could be provided by supplying a discriminant parameter to the object. This method was chosen because it had none of the associated disadvantages of the other options. Furthermore, it offered the following advantages:

- A direct correlation would exist between declarations and filename associations, i.e. for every declaration there would be exactly one filename supplied and the name would always be supplied at the same time as the declaration.
- Assignment between containers referencing different files would automatically raise a constraint error thus preventing loss of the assigned filename for the target and enforcing the use of a "Copy" routine for transferring data between containers.

More details of the implementation of the discriminant can be found in the chapter "Persistent_Container" (page 29). Given this method of specifying the filename, the behaviour required for persistent containers (additional to that exhibited by their volatile equivalents) was identified as follows:

5.3.3 Initialization

On declaration of a persistent container the supplied filename should be checked. If the name is null then the container should behave in an identical manner to the equivalent volatile container otherwise the behaviour should depend on whether a file having the given name already exists. If such a file is found to exist then the data should be loaded from that file into the container otherwise a new, empty file should be created having the supplied name.

5.3.4 Determining filename

A function should be provided allowing client programs to determine the assigned filename:

```
FUNCTION Assigned_Name(Obj : Persistent_Container) RETURN String;
```

5.3.5 Finalization

Before a persistent container is destroyed within a program, its volatile data elements in memory should be saved to the associated disk file.

5.3.6 Data copying

Assignment, using ":=", between two objects having different discriminants raises a constraint error. It will therefore be impossible to use this form of assignment to copy data between persistent containers referencing different files. However, since it is necessary to allow for data copying, an alternative method of assignment should be provided. The proposed solution was to provide a routine, declared as follows, to copy the data elements only from one container to another without affecting the associated filenames:

```
PROCEDURE Copy(From : IN Persistent_Container;
               To : IN OUT Persistent_Container);
```

5.4 External containers

External containers were to be implemented using external disk files as primary storage − data items being loaded into memory only accessed.

The implementation of an external container could not therefore be based on the existing volatile containers (which use main memory) but the behaviour and interface to the container was to be as close as possible, i.e. the difference in implementation between external and other types of containers was to be invisible to the user. In order to achieve this, it was decided that external containers should inherit the interface of the equivalent volatile container but not the implementation (e.g. an external stack would be derived from the existing abstract "Stack" tagged type rather than from the concrete "Bounded_Stack" or "Dynamic_Stack" tagged types etc.).

The interface to the persistent aspects of external containers were to be similar to those of persistent containers, i.e. the same facilities were to be created for associating and determining the associated filename and for copying data between containers. One exception, however, was identified, as follows:

An external container assigned a null filename should still make use of an external file to hold the data elements even though the data elements are not to be retained after the lifetime of the variable. Supplying a null filename must not result in the container using main memory rather than an external file since an external container may be used for the purpose of holding a larger volume of data than could be held in main memory. In practice, this would prove easy to implement since the default behaviour of Ada 95 when dealing with null filenames is as required. When an attempt is made in Ada 95 to open a file having a null name, a temporary scratch file is automatically assumed (Ada Reference Manual 1995, ch. A.8.2). This would provide an ideal solution as it would mean that the data elements will be held in external storage (as required) but that the external file will be removed (and thus consume no disk space) as soon as the program terminates.

5.5 Style

A consistent coding style was to be adopted throughout the project. The style was to allow for readable code both on screen and when printed on a line printer. For this reason, the maximum line length was limited to eighty characters and keywords were to be in uppercase characters for easy identification. Horizontal and vertical white spacing was to be inserted consistently and comments were to be adequate but not excessive.

In order to help ensure a consistent approach to style, it was decided to make use of the "Ada−mode" feature of the Emacs text editor. This would automate the capitalisation of characters and the indentation of lines.

Other matters of style were consistently maintained as encouraged by the Computer Science department of Aston university. For example, the mode of parameters (i.e. "IN", "OUT" or "IN OUT") was always specified in procedure declarations but not in function declarations (where the mode "IN" is mandatory and hence always assumed).

6 Implementation

A high−level description of the system implemented and low−level details of noteworthy aspects along with testing methods and their outcomes.

6.1 Data items

As discussed in the "Design" section (see "Data items", page 12), it was required that the non−volatile containers were compatible with all non−limited definite data items. It was therefore necessary to ensure that all such items could be stored in persistent files.

The standard Ada packages "Ada.Sequential_IO" and "Ada.Direct_IO" were to be used to manage the storage of data items in files. The element types supported by these packages was determined by the packages' generic parameters (Barnes 1996, pp. 504 & 508):

```
GENERIC
   TYPE Element_Type(<>) IS PRIVATE;
PACKAGE Ada.Sequential IS
...

GENERIC
   TYPE Element_Type IS PRIVATE;
PACKAGE Ada.Direct_IO IS
...
```

From these definitions, it was deduced that both of these parameter types were flexible enough to handle any definite type for which assignment and equality were predefined (Barnes 1996, pp. 370−371). The "Element_Type" parameters of "Ada.Sequential_IO" and "Ada.Direct_IO" are both therefore capable of handling any data item supported by the containers of the Booch Components. Using these file handling packages, it was therefore deemed possible to implement non−volatile versions of the Booch Components that would make no further restrictions as to the allowable data items, i.e. the persistent components would be capable of handling even complex dynamic data structures such as bounded strings (see "Bounded_String", page 19). Unfortunately a complication was identified for those data items based on data structures using access values such as unbounded strings (see "Unbounded_String", page 19).

6.1.1 Bounded_String

Bounded strings (as defined in "Ada.Strings.Bounded") are an example of a complex but, as shown by the following type declaration (Ada Reference Manual 1995, ch. A.4.4), non−limited definite type:

```
      TYPE Bounded_String IS PRIVATE;
```

Robert Leif suggested that an example of the use of the non−volatile containers with bounded strings should be included (see "Robert Leif reply", page 86). To this end, a spell checker program was written which was implemented using a persistent unbounded set of bounded strings (see "Spell checker", page 45).

6.1.2 Unbounded_String

Unbounded strings (as defined in "Ada.Strings.Unbounded"), like bounded strings, are implemented as a non−limited definite type (Ada Reference Manual 1995, ch. A.4.5):

```
      TYPE Unbounded_String IS PRIVATE;
```

Because of this, it was expected that containers having unbounded string data items would work correctly and, for the volatile containers, that was so. Unfortunately tests showed that unbounded strings (and other types making use of access values) were not suitable for use as data items for non−volatile containers. The reason for this is discussed below:

Unbounded strings are implemented as a definite type that contains an access value that references dynamically allocated data actually representing the string. When such a data item is held in a persistent container, the definite data item is stored in the associated file but the heap memory referenced by the access value is not. Thus, when the program ends the actual data in the heap containing the string is lost.

Implementing persistent containers that would work correctly with types using access pointers would have significantly increased the complexity of the project and hence was accepted as a limitation. Justification of this decision follows:

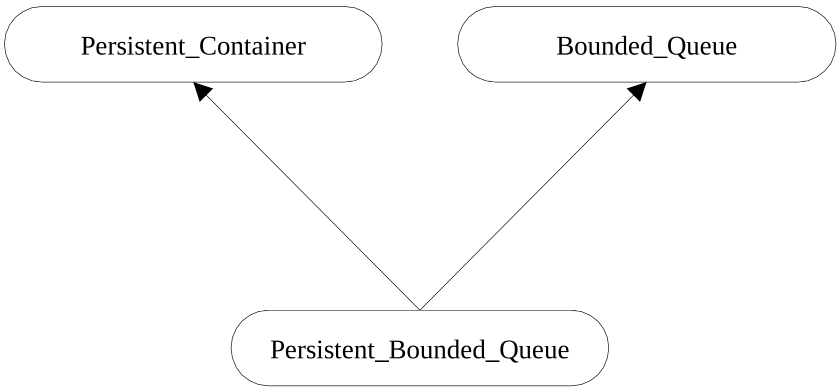
- The built in Ada file handling routines for direct and sequential IO have the same limitation. They store access values without automatically storing the data referenced by the access value.

- The most general solution to the problem might involve implementing an alternative storage pool that effectively implements a persistent heap (Card 1997). Such a solution would be implemented independently of the Booch Components.

6.2 Multiple inheritance

Each non−volatile container was to be derived from two parent tagged types. For persistent containers (such as "Persistent_Bounded_Queue") one parent would be "Persistent_Container" (an abstract container providing the functionality common to persistence) and the other would be the existing volatile container (e.g. "Bounded_Queue"). For external containers (such as "External_Queue") one parent would be "External_Container" (implementing the functionality common to external containers) and the other would be the existing interface (e.g. "Queue").

Figure 2 − example of multiple inheritance required.



Since no direct support for multiple inheritance was available in Ada 95, an alternative method of achieving the same effect was required. Various methods were considered as follows:

6.2.1 Support packages

The first solution considered was to inherit from just one parent. The persistent or external functionality would be implemented using calls to routines in a support library rather than by inheriting from a "Persistent_Container" or "External_Container" parent. This solution proved unsatisfactory as it resulted in a great deal of code duplication, such as the inclusion of a separately coded "File_Name" attribute in each non−volatile container definition.

6.2.2 Record wrapper

The second solution considered was to make each new container a composite record type as suggested by Smith (1996, p. 172). This proposal suggested that a separate attribute of the record should be used to hold an instantiation of each parents, e.g. an external queue would have been implemented as:

```
TYPE External_Queue IS RECORD
   Q : Queue;
   E : External_Container;
END RECORD;
```

This implementation proved unworkable for this situation because "Persistent_Container" and "External_Container" were both abstract and hence could not be instantiated as attributes of a record.

6.2.3 Mixin inheritance

Mixin inheritance as described by Barbey *et al.* (1993, ch. 11.2) and Barnes (1996, p. 452) was chosen as the solution to the multiple inheritance problem to be implemented. This method makes use of the generic features of the Ada 95 programming language allowing one sub−class to be extended with another. There follows a description of the mixin inheritance implementation used in this project:

6.2.3.1 Persistent containers

In order to implement persistent containers using mixin inheritance the abstract "Persistent_Container" tagged type was coded as an extension of a generic container (the container being supplied as the generic parameter "Structure"):

```
GENERIC
   TYPE Structure IS ABSTRACT NEW Container WITH PRIVATE;
PACKAGE BC.Containers.Persistent IS
   TYPE Persistent_Container IS ABSTRACT NEW Structure WITH PRIVATE;
   ...
END BC.Containers.Persistent;
```

Each persistent container creates an instance of "Persistent_Container" with the volatile version of the container supplied as the generic parameter, e.g.

```
WITH BC.Containers.Persistent;

GENERIC
PACKAGE BC.Containers.Queues.Bounded.Persistent IS

   PACKAGE Persistence IS NEW BC.Containers.Persistent(Bounded_Queue);
   USE Persistence;
   TYPE Persistent_Bounded_Queue IS NEW Persistent_Container WITH PRIVATE;

   ...

END BC.Containers.Queues.Bounded.Persistent;
```

The above example illustrates how "Persistent_Bounded_Queue" was implemented as a descendant of both "Persistent_Container" and "Bounded_Queue". The remaining persistent containers were implemented using the same technique.

6.2.3.2 External containers

The multiple inheritance required to implement external containers was similar to that of persistent containers. Each external container was to be derived from "External_Container" and a type defining the interface (e.g. "Queue"). The only difference between the multiple inheritance required for external containers and that for persistent containers (as described in the previous section) was that both, rather than just one, of the parent types were to be abstract. This posed no problems since the generic "Structure" parameter of "External_Container" had been declared as:

```
      TYPE Structure IS ABSTRACT NEW Container WITH PRIVATE;
```

The inclusion of "ABSTRACT" in this declaration enabled abstract types to be supplied as the generic parameter.

6.2.3.3 Summary of mixin inheritance

The use of mixin inheritance proved hugely beneficial in terms of implementing both persistent and external containers with a minimum of code duplication. It allowed all the common code supporting persistent and external structures to be factored out into abstract types. Unfortunately mixin inheritance did also introduce a small penalty (discussed below) in addition to adding to the complexity of the components:

The use of mixin inheritance prevented easy extension of the "Create" routines for the persistent dynamic containers. For example, if "Persistent_Dynamic_Queue" had been an extension of just "Dynamic_Queue", rather than an extension of both "Dynamic_Queue" and "Persistent_Container", then the "Create" routine for "Persistent_Dynamic_Queue" could have been implemented using straightforward downward conversion of objects (Barnes 1996, p. 257), i.e. it could simply have returned the result of the "Create" function for "Dynamic_Queue" along with any additional fields specific to "Persistent_Dynamic_Queue".

Alternative solutions to the implementation of the "Create" routines using temporary objects of type "Persistent_Container" were ruled out because it was an abstract type. As a consequence it was necessary to implement the routines using a small amount of code duplication from the volatile variants. The appendices include an extract from "BC.Containers.Queues.Dynamic.Persistent" that show the implementation of the "Create" function for "Persistent_Dynamic_Queue" (see "Create function for persistent dynamic containers", page 79).

6.3 Equality testing

In order that the equality test function ("=") for the non−volatile containers would compare only the data in the container and not additional attributes (such as the filename), it was necessary to override the inherited or assumed equality test for both persistent and external containers by specifying a new one.

Persistent containers were coded as derivations from the volatile containers and, hence, normal inheritance rules would have dictated that they would have inherited the parent equality test function and thus worked correctly. As Barnes (1996, p. 276) describes, however, it is unusual to required that the equality test should ignore new attributes of an extended type and, hence, Ada 95, by default, does not behave as required for this case. In this instance, where the new attributes were to be ignored, it was therefore necessary to provide a new function to override the assumed equality test for each persistent container. The required functionality was achieved using a dispatching call (Barbey *et al*., 1993) to the parent's equality test. As an example, the equality test for persistent bounded queue has been included in the appendices (see "Bounded persistent queue equality test", page 78).

6.4 Memory mapping

As mentioned elsewhere in this report (see "Persistent data", page 14, "Lists", page 35 and "Graphs", page 51), the use of memory mapping (Hall 1997) was considered as a method of implementing persistent data structures. This would have been achieved by making use of Florist (Baker 1999) − a set of packages providing Ada 95 POSIX bindings (Gallmeister 1995, ch. 4). The possibility of implementing persistence using this approach was investigated as follows:

All the volatile containers were coded to access heap memory using the support routines from the package "BC.Support.Managed_Storage". This package implements a storage pool as describe by Barnes (1996), Card (1997), the Ada 95 Rationale (1995) and the Ada Reference Manual (1995). The investigation therefore centred around coding a new storage pool manager that would implement persistent storage using a memory mapped file. Each of the persistent containers would then have been implemented by creating instantiations of the existing volatile components with the new storage pool.

After review, this technique was abolished for the following reasons:

- Retrieval of persistent data: The storage manager would allow automatic saving of the data but additional routines would still be required in the containers in order to retrieve the data since the existing implementations do not expect a persistent storage pool and therefore initialise values.
- Dynamic re−sizing: The memory mapping approach did not lend itself well to dynamic data structures. Increasing the size of a memory mapped file would have introduced a need to un−map and then re−map the memory from the file. No method of doing this could be determined that would guarantee that the new mapping would access the same memory location as the previous mapping and, hence, there was no guarantee that existing pointer values in the data would continue to reference the correct address.

  Further potential problems identified using memory mapping for dynamically re−sizeable containers included fragmentation. Memory freed from anywhere other than the end of the file could not be reclaimed because it would result in holes in the file. Files (and hence the mapped memory) would thus have never reduced in size below their maximum used size. Furthermore, if allocations were not always of the same size, the file could have grown indefinitely as fragmentation resulted in gaps too small to be useful.

- Not all users of the Booch Components would have the Florist packages and, hence, the Florist library would have had to be distributed and installed along with the Booch Components.
- Use of Florist would have defeated many of the advantages of using Ada. The Florist libraries contain many platform dependencies, for example restricting the compiler to GNAT, and make for more complicated and less readable code.

6.5 Filenames

Both persistent and external containers were implemented using references to external files. Any such file, on a POSIX system, is required to be provided with a filename in order that it can be identified. In order to support this, a private abstract data type was implemented in the package "BC.Support.File".

The "Filename" ADT, as produced during the course of this project, was intentionally kept basic. The ADT was made to represent filenames as an unbounded strings and to provide routines to convert between strings and filenames. The intention was that users would tailor the package to their particular systems so that the use of invalid filenames could be handled more conveniently, for example by raising suitable exceptions. This could not have been implemented as part of the project because a general solution was not possible given the variation of rules for determining the validity of filenames between different systems.

6.6 Persistent_Container

An abstract tagged type ("Persistent_Container") was implemented as a parent type from which all the concrete persistent containers were derived. This code was used to factor out code common to all persistent containers.

In order to ensure that any persistent container derived from "Persistent_Container" specified how to load and save to and from a persistent file, two abstract procedures were declared as follows:

```
      PROCEDURE Load_Data(Obj : IN OUT Persistent_Container) IS ABSTRACT;
      PROCEDURE Save_Data(Obj : IN Persistent_Container) IS ABSTRACT;
```

Given the assurance that every concrete persistent container would provide load and save routines, "Persistent_Container" was able to automate the loading of persistent data from file at object creation and the saving of persistent data to file at object destruction. This was achieved through the use of the "Initialize" and "Finalize" routines inherited from the abstract tagged type "Ada.Finalization.Controlled". These routines are automatically called immediately after objects are created and just before objects are destroyed respectively (Barnes 1996).

The abstract "Persistent_Container" tagged type was also used to handle the filenames associated with persistent containers. This was achieved through the use of an access discriminant of type "Filename" (see "Filenames", page 28). A function was also defined in order to determine the filename associated with a container:

```
      FUNCTION Assigned_Name(Obj : Persistent_Container) RETURN String;
```

The reasoning behind using a discriminant for specifying the filename was discussed above (see "Persistent data", page 14). The reason for using an access discriminant was that "Filename" was not discrete and, thus, could not be used directly as a discriminant (Barnes 1996, p. 335).

6.7 External_Container

An abstract tagged type ("External_Container") was implemented as a parent type from which all the concrete external containers were derived. This was used to factor out the code common to all external containers in the same fashion that "Persistent_Container" was implemented for persistent containers (see previous section). Consequently, there are many similarities between "External_Container" and "Persistent_Container".

One commonality between "External_Container" and "Persistent_Container" is the manner in which the associated filename is handled. An access discriminant (of type "Filename") was implemented (see "Filenames", page 28) as was a function to determine the assigned filename:

```
FUNCTION Assigned_Name(Obj : External_Container) RETURN String;
```

Like "Persistent_Container", "External_Container" was also implemented using the "Initialize" and "Finalize" routines inherited from "Ada.Finalization.Controlled". External containers, however, were not required to automatically load and save data to a persistent file at creation and destruction because, by definition, they were to continually maintain their data in a file throughout their lifetimes. The "Initialize" and "Finalize" procedures of "External_Container" are therefore used only to establish and to end the association between the container and its external file rather than to actually handle the loading and saving of data. At initialization, an external container creates a "File_Type" variable that accesses either an existing file holding persistent data or, if no existing file is present, a newly created file. At finalization, the file is closed and the "File_Type" variable destroyed to free up memory.

An access value was used to implement the "File_Type" variable rather than providing a direct "File_Type" attribute of "External_Container" for the following reasons:

- "File_Type", as defined in "Ada.Sequential_IO" and "Ada.Direct_IO", is limited (Barnes 1996, p. 504) but "External_Container" is not. The use of an access value allowed the limited attribute to be incorporated into the non−limited record.
- Use of an access value provided a mechanism for clients of "External_Container" to access the "File_Type" variable directly. Some operations, such as deleting files, require that the "File_Type" argument supplied is a variable because the mode of the parameter is "IN OUT". "External_Container" was therefore coded to provide access to the "File_Type"

access value using the function "File_Ptr_Of" thus providing direct access to the variable. Deletion could then be performed as follows (assuming "Obj" to be an instance of an "External_Container"):

```
    Delete(File_Ptr_Of(Obj).ALL);
```

6.8 External stacks

6.8.1 Inverse ordering

Since data items for the existing volatile stacks are held in random access main memory, they can be inserted and removed with equal efficiently from either end of the data structure. External stacks, however, were required to hold data item in a disk file and, due to the nature of such files, were able to append data items to the end of a file far more efficiently than inserting items at the front. Because of this, it was decided to store the data items of external stacks such that the top of stack was represented by the end of the file. Items would then be pushed on to the stack by appending to the file rather than inserting at the front of the file.

As a consequence of using this ordering of data items, it was necessary to take care with the implementation of the iterator functions. It was necessary to ensure that the items of external stacks were iterated in the same order as other stacks so that, for example, the equality test worked correctly between external and other stacks − the equality test had already been implemented in the existing Booch Components using an iterator to traverse each stack comparing each data item in turn.

6.8.2 Two versions − small and fast

As described above, the most efficient order was chosen for storing data items in the file in order to maximise the efficiency of pushing items onto the stack. Considering the efficiency of popping items off the stack however resulted in a more complicated decision. The two options were identified as follows:

1. The last item could be removed by truncating the file. Using the standard Ada 95 routines this would be a slow procedure requiring the use of a temporary scratch file but it would results in the most efficient possible use of disk space.
2. The last item could be removed by decrementing a variable indicating the count of the number of items in the stack. This would be a fast procedure but would result in wasted disk space.

The decision was made to implement both of these options. Two versions of external stack were coded − "Small_External_Stack" using the space efficient method and "Fast_External_Stack" using the speed efficient method. In order to eliminate code duplication between "Small_External_Stack" and "Fast_External_Stack", an abstract tagged type ("External_Stack") was implemented incorporating the common code. This tagged type became a parent from which both concrete external stacks were derived.

6.8.2.1 Small_External_Stack

The implementation of "Small_External_Stack" was straightforward in comparison to "Fast_External_Stack". The only complication being the necessary use of a temporary scratch file due to the lack of a truncate routine in Ada. The file was therefore effectively truncated by saving the data items in a scratch file, deleting the original file and then re−creating it. The algorithm that achieved this is included in the appendices (see "Small_External_Stack pop procedure", page 80).

6.8.2.2 Fast_External_Stack

As described above, fast external stacks were not required to truncate the external file when data items were popped off. As a result, space occupied in the file by the popped item would not be reclaimed, however, pushing another item back onto the stack would reuse any previously occupied space. The space used by the file would therefore never become any greater than the maximum size reached by the stack.

In order to allow for the reloading of the persistent data of fast external stacks it was necessary to modify the external file. Without doing this it would have been impossible to determine which data items were referenced and which un−referenced. It was therefore decided to truncate the file to eliminate the un−referenced data items at finalization. Although a slow procedure, this would be performed only once per object lifetime and would also provide the additional benefit of reducing the file size thus freeing up unused space.

6.9 Rings

Unlike the other Booch Components containers, rings had been implemented with attributes in addition to the actual data items − each ring also has natural values indicating the positions of its "top" and "mark". Thus, for persistent rings to work correctly, both the data items and the additional components had to be stored in the associated file.

Normal Ada files, as handled by "Ada.Direct_IO" and "Ada.Sequential_IO", were not suitable for implementing persistent rings because two different types of data item could be required to be stored in the persistent file − two natural values for "top" and "mark" followed by an arbitrary number of data items. It was therefore necessary to implement the persistent behaviour of rings using streams (Ada Reference Manual 1995, ch. 13.13 and Barnes 1996, p. 519) allowing the creating of heterogeneous files having elements of different types.

The following differences were required for persistent rings, compared with other persistent containers, in order to make use of streams:

1. The "Save_Data" and "Load_Data" routines of persistent rings were required to store and retrieve the two natural values at the start of the stream before processing the data elements in the remainder of the stream. The code of these routines is included in the appendices (see "Persistent ring streams", page 70).
2. Two additional routines ("Read_Data_Stream" and "Append_Data_Stream") were included in the support package "BC.Support.Unbounded.Persistent" (see "Support packages", page 40).

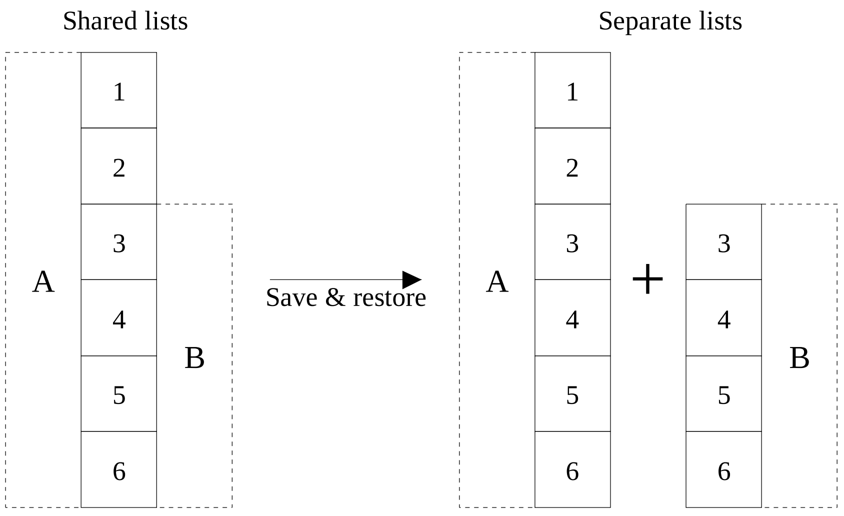
6.10 Lists

The volatile versions of lists allow data elements to be shared between several list containers. Thus, for example, one list may form the tail of another list. This was implemented for volatile containers using access values from one list directly referencing data elements of other lists − possible because all the data elements of volatile lists are held in the same storage pool.

The persistent versions of lists were based directly on the volatile versions and, hence, retained the behaviour allowing sharing of data elements. Unfortunately, due to the method of saving the data elements in a separate files for each container, no feasible method of retaining the information relating to the sharing of elements was found. Consequently, although the sharing of data elements works for persistent lists during the lifetimes of the objects, when objects are re−established each has its own duplicate copy of the previously shared elements.

This behaviour is accepted as a limitation of persistent lists but remains due to the complexity that would have been introduced in order to overcome it. The use of memory mapping was considered as one potential alternative solution that would have eliminated this restriction but proved infeasible (see "Memory mapping", page 26).

Figure 3 − Sharing of lists.



6.11 Iterators

No effort was required for the implementation of iterators for persistent containers since the inherited iterator functions of the volatile containers worked correctly.

Providing iterators for the external containers, however, was far more challenging. Given the use of files for storing the contents of external containers, the data could not be referenced using access values. The iterator routines of the volatile containers were therefore not appropriate for such containers.

An initial solution to this problem was considered based on the principle of making a temporary heap allocation for the data item currently referenced by the iterator and using an access value to that area of heap. This solution proved infeasible due to the following problems:

- The "Storage_Size" attribute for "Item_Ptr" had been set to zero in the package "BC.Containers" and hence heap space could not be allocated for data items.
- No method could be found to determine when the allocated heap was no longer needed allowing the space to be de−allocated and any changes made to the data saved back to the external storage.

An alternative method of achieving iteration for external containers was therefore required. The final method chosen was implemented as follows:

1. A new iterator type (i.e. "External_Queue_Iterator" and "External_Stack_Iterator") was defined for each external container. This was required in order to distinguish external iterators from other iterators and thus allow different iteration routines to be applied to external containers.
2. The function "Current_Item", returning an access value to the item currently referenced by an iterator, was overridden for external containers. It was modified so that it would always raise a suitable exception (e.g. "Not_Available_For_External_Queues") thus aiding debugging of any code that continued to use the invalid routine.
3. The function "Current_Item", returning a copy of the current item referenced by an iterator, was also overridden for external containers. It was modified so that it did not rely of the existence of the other "Current_Item" routine returning an access value.

4. The "New_Iterator" functions produced for each external container were implemented as those for volatile containers but using the iterator specific to the external container rather than the standard iterator.
5. A replacement for the "Access_Current_Item" procedure (from the package "BC.Containers") was coded in each package implementing an external container. The replacements were designed to work with the external containers in addition to the volatile and persistent containers supported by the old version. This was achieved by incorporating a test to determine whether the iterator being applied was specific to the external container class (Barnes 1996, p.272). If not it would use the "Current_Item" routine compatible with volatile and persistent containers, otherwise it would work in a manner specific for the external container − i.e. explicitly load a copy of required data item into memory, perform the required access and then explicitly write the modified copy of the data item back to the external file.
6. Replacement for the procedures "Modify", "Modify_With_In_Param" and "Modify_With_In_Out_Param" (from "BC.Containers") were coded in each package implementing an external container. The code of the replacements was identical to that of the corresponding existing routine but it was necessary to duplicate the code in the alternative package so that it would work using the alternative "Access_Current_Item" routine instead of the original one in "BC.Containers".

This implementation of external iterators resulted in the potential for several copies of "Access_Current_Item" and "Modify" being visible to client programs − all would work with volatile and persistent containers but only one would work with any particular external container. It is therefore essential that client programs call the correct procedures (otherwise "Not_Available_For_External_*xxx*" exceptions will be raised at run−time). This could be achieved with the inclusion of a RENAMES clause (Barnes 1996, p. 248) such as:

```
GENERIC PROCEDURE Modify RENAMES QE.Modify;
GENERIC PROCEDURE Access_Current_Item RENAMES QE.Access_Current_Item;
```

Where "QE" is the generic instantiation of "BC.Containers.Queues.External". This gives the "Modify" and "Access_Current_Item" routines of "QE" priority over any other overloaded versions.

6.12 Process_Front & Process_Top

Coding the "Process_Front" and "Process_Top" routines for external containers introduced problems similar to those introduced by external iterators (see "Iterators", page 36). The problems were therefore overcome in a similar manner, i.e. by replacing the problematic procedures in "BC.Containers.Queues" and "BC.Containers.Stacks" with alternatives in "BC.Containers.Queues.External" and "BC.Containers.Stacks.External".

Like the iteration routines, the alternative versions of these procedures were coded in such a way as to ensure that they would work with volatile and persistent containers as well the particular external container being considered. Also like the iteration routines, the solution requires that client programs use the correct version of the procedure, for example by including a RENAMES clause such as:

```
GENERIC PROCEDURE Process_Front RENAMES QE.Process_Front;
```

6.13 Exceptions

As far as possible the exception mechanism of the new packages was kept consistent with that of the existing Booch Components. The external containers were therefore coded to raise common exceptions (such as container underflow) in the same manner as the volatile containers, i.e. by calling routines from "BC.Support.Exceptions". See "External exceptions" (page 72) for an example from the external queue implementation. Since persistent containers were coded by inheriting behaviour from volatile containers, they automatically acquired the appropriate method of handling common exceptions.

In addition to the exceptions common to both volatile and non−volatile containers, however, the new containers introduced a need for a number of new exceptions. External containers, for example, required an exception to prevent the use of the incorrect version of "Access_Current_Item" (see "Iterators", page 36). This introduced a small inconsistency in the implementation as follows:

New exceptions introduced specifically for non−volatile containers were declared in the specifications of the packages in which they were referenced rather than in the package "BC" where all existing Booch Component's exceptions were declared. The justification for this being that the inconsistency was deemed preferable to making changes to the existing file which could have affected existing programs using the Booch Components (see "Design", page 10).

The use of exceptions was also considered with regards to the use of invalid filenames for persistent and external containers. It was decided, however, that the only viable solution was to rely on the built−in system exceptions for this error because the rules for governing the validity of filenames would not be consistent between different systems. A package handling filenames (see "Filenames", page 28) was therefore coded with the intention that users would configure it to their own system requirements if necessary.

The file "exception_test.adb" (see "Exception_Test", page 73) demonstrates some exceptions of the components.

6.14 Support packages

The volatile containers of the Booch Components were implemented from common data structures. Support for these data structures was provided by the following support libraries:

- **BC.Support.Bounded** − providing an array implementation for bounded containers.
- **BC.Support.Unbounded** − providing a linked list implementation for unbounded containers.
- **BC.Support.Dynamic** − providing a re−sizeable array implementation for dynamic containers.
- **BC.Support.Hash_Tables** − providing a hash table implementation (in turn using bounded, dynamic or unbounded structures) for containers where efficient random access is important such as sets and bags.
- **BC.Support.Nodes** − providing re−sizeable implementations of linked data structures such as lists and graphs.

A child package of each of these support packages was coded to provide support for persistent containers. Each persistent support package provided routines to save and load the relevant data structure to and from a file. Persistent containers were then able to make use of these routines thus eliminating the need for duplicate code between containers based on the same underlying structure.

The body of the package "BC.Support.Bounded.Persistent" is included in the appendices (page 75). The "Read_Data_File" and "Write_Data_File" routines of this package are almost identical to those in the other persistent support packages ("BC.Support.Dynamic.Persistent" and "BC.Support.Unbounded.Persistent") because the interfaces to the ADTs "Bnd_Node", "Dyn_Node" and "Unb_Node" are all very similar. If "Bnd_Node", "Dyn_Node" and "Unb_Node" had been implemented as tagged types derived from a common parent then there would have been the possibility of factoring out the common code to reduce duplication but this was a limitation imposed by the existing implementation.

As mentioned above (see "Rings", page 34), additional routines have been supplied in "BC.Support.Unbounded.Persistent". "Read_Data_Stream" and "Append_Data_Stream" were provided to allow data items to be read from and appended to end of a file stream.

6.15 System compatibility

As far as possible, the implementation of the new containers was kept as system independent as possible. Following is a discussion of various considerations made, during the development of the project, relating to system compatibility:

6.15.1 Compiler

The only Ada compiler available for use during the development and testing of the components was the GNU Ada Translator (GNAT). This is therefore the only compiler with which the components have been tested but there is no known reason why other Ada compilers should not work. The GNAT compiler itself has been implemented on a wide range of systems including DOS, Windows, Unix, Linux and Macintosh OS.

Version 3.11 of the GNAT compiler would not compile those packages implementing persistent containers based on hash tables (i.e. bags, map and sets) (see "GNAT bug", page 50). This is believed to be due to a bug in the particular version of the compiler since version 3.12 does compile the same code successfully.

6.15.2 File system

The files making up the Booch Components have been implemented using the GNAT default filenames for systems with long filename support. This means that problems will arise on systems that have a short limit on the length of filenames. The justification for this is that most modern operating systems support filenames of adequate length for this naming convention and, in any case, it is easy to shorten the names of the files for non−compliant systems using the "gnatchop" utility that forms part of the GNAT system.

The filename ADT (as implemented in the package "BC.Support.File") has been kept as generic as possible (see "Filenames", page 28) so as to allow any filename that is valid on any system. It has been left up to the user of the Booch Components to implement any filename validity checks deemed necessary.

6.15.3 Existing components version

Due to variations between each version of the Booch Components, it was necessary to match the code produced during the course of this project with a particular version of the components.

When coding began on the project, the most recent version of the Booch Components was 19991031 and, hence, that version was used. Part way through the project, the next version (20000103) of the Components was released and small modifications were made to my code to maintain compatibility with the newer version. Later on during the project, a third version (20000219) of the components was released. Due to the late stage during the project that this update occurred and the problem that it would have introduced (see following) it was decided to continue development matched to version 20000103 of the Booch Components.

6.15.3.1 20000219 problem

The version 20000219 update of the Booch Components changed the package "BC.Support.Hash_Tables" so that it consisted of three sub−packages ("Item_Signature", "Value_Signature" and "Tables"). This introduced a problem since extension of these sub−packages would not have been possible without modifying the existing file. The decision to stick with the previous version of the Components (20000103) was therefore made with the following justification:

Two options were identified for allowing the persistent containers to work with the latest version of the Booch Components. Both options require changes to existing files and should therefore be performed only by the official maintainer of the components:

1. Modification of the implementation of "BC.Support.Hash_Tables" so as to use child packages (implemented in separate compilation units) rather than sub−packages within the same unit. This would enable extension of the code without further modification to the existing files.
2. Modification of the implementation of "BC.Support.Hash_Tables" so as to incorporate the extensions required to implement persistence.

The first point above raises a general implementation issue having importance regarding system extensibility. Packages can be nested as sub−packages, such as:

```
PACKAGE A IS
   PACKAGE B IS
      ...
   END B;
   ...
END A;
```

This structure however prevents the extension of the nested package without editing the existing compilation unit, thus, requiring recompilation of existing code potentially affecting existing client units.

If a package is therefore to be coded in a manner allowing for safe extension then the use of nested sub−packages should be avoided. An alternative implementation could make use of child packages (Barnes 1996, pp. 238−242). The above example could therefore have been implemented as follows in order to allow for safer extension:

```
PACKAGE A IS
   ...
END A;

PACKAGE A.B IS
   ...
END A.B;
```

This implementation would allow the child package to be extended without editing the existing file (i.e. by coding a new child package "A.B.C"). Additionally, this implementation would provide the advantage of reducing development times by allowing the two packages to be compiled separately (i.e. making no requirement that both packages be re−compiled after a change to just one of them).

6.16 Testing

6.16.1 Individual tests

Simple command line programs to test the primitive features of each non−volatile container were implemented in a similar to style to the test programs for the volatile containers. The test programs for the non−volatile containers were closely related to those of the volatile containers with the following notable differences:

- **Persistence tests** − At the start of each test program any persistent data found in the containers is tested. If no persistent data is found (i.e. the containers are empty) then it is assumed to be the first run of the test program and an appropriate message is given. If persistent data is found then the data is examined and a message confirms whether or not the found data matches what was expected. At the end of each test program specific data is stored in the containers so that the program knows what persistent data to expect in subsequent runs.
- **Assignments** − Assignments as used throughout many of the original test programs are not valid for persistent or external containers and hence they were replaced by calls to the "Copy" routine (see "Data copying", page 15).
- **Support package** − Many of the routines in the test programs were common for each container and hence were factored out into a support package "Test_Support".

Example output produced by these test programs is included in "Appendix 4 − Test program output" (page 82).

6.16.2 Additional tests

A number of additional test programs were coded to test features of the new Components that were not specific to a particular container:

- Two test programs ("Are_Equal_Queue_Test" and "Are_Equal_Stack_Test") were coded to ensure that equality comparison worked correctly. "Are_Equal_Queue_Test", for example, checks equality comparisons between volatile, persistent and external queues.
- A further test program (see "Exception_Test", page 73) was coded to check the exceptions raised by persistent and external containers in certain error situations.

6.16.3 Spell checker

In addition to the programs written to specifically test particular features of the components, a simple command−line spell checker program was written as a demonstration of a "real" application of persistent containers.

The spell checker program was coded using a "Dictionary" abstract data type, provided by the package "Spell_Check_Support" and implemented as a persistent unbounded set. Dictionary ADTs are able to store words of type "Dict_Word" which is actually an instantiation of "Bounded_String" and, consequently, serves to satisfy Robert Leif's request that an example of the use of the containers with bounded strings is supplied. The particular instantiation of "Bounded_String" used has a maximum of thirty characters so the dictionary is limited to words not exceeding that length. For reasons discussed above (see "Unbounded_String", page 19) the use of "Unbounded_String" was not possible with a persistent container.

The spell checker implementation actually contains two dictionary ADTs − one with an assigned filename ("dictionary.dat") is used as a persistent store for the words that are to be remembered permanently. The other, with a null filename (""), is used to store words which the user has chosen to ignore for the remainder of the particular session. Because a null filename is used, the persistent unbounded set behaves just like a volatile unbounded set and hence is not saved to a persistent file at the end of its lifetime.

The Booch Components implements sets as (chained) hash tables and hence, in order to implement the dictionary as a persistent unbounded set, it was necessary to specify the number of buckets (chains) to be used and to provide a hash function for the "Dict_Word" type. The number of buckets for the hash table was selected to be sixty−three (a prime number) and a simple hash function (see "Spell checker hashing function", page 77) returning the sum of the ordinal representation of the characters of the string was chosen. Use of a larger number of buckets and / or an improved hashing algorithm may have made a significant improvement to the performance of the spell checker when handling large dictionaries but the selected combination was sufficient for the purpose of demonstrating the application. The reader is referred to Harrison (1989, ch. 9) for more details on chained hash tables and improved hashing functions.

The case of the characters of words in the dictionary is maintained − i.e. character are stored in the same case as they are supplied. When checking words, the spell checker requires that characters stored in uppercase in the dictionary are matched with uppercase letters in the document but makes no similar requirement for lowercase letters. This provides the following desirable features:

- If "mark" is stored in the dictionary then both "mark" and "Mark" will be accepted as correctly spelt words.
- If "Robert" is stored in the dictionary then "Robert" but not "robert" will be accepted as a correctly spelt word.
- Duplicate words having different case letters can be stored in the dictionary, so if "Mark" is in the dictionary then "mark" can still be inserted to allow both "Mark" and "mark" to be accepted.

Instructions detailing how to use the spell checker are provided in the appendices (see "Spell checker", page 61).

7 Evaluation

An explanation of the evaluation methods applied to the end−product and a discussion of the success of the project in terms of how well the original objectives were realised and the reliability and usability of the end−product.

7.1 Objectives met

The goals of the project were as follows:

1. Extending knowledge of the object−oriented features available in Ada 95.
2. Coding persistent versions of the existing Booch Components' containers.
3. Coding external versions of the existing Booch Components' containers. This was given a lower priority for the reasons discussed in the "Context" (page 5).
4. Coding additional volatile containers to those already existing in the Booch Components. This requirement was included in the initial "Final Year Project Definition" (page i) but dropped from the "Requirements" (page 8) after initial investigation showed that work load of performing this task along with the other requirements would have been unrealistic given the time span allowed.

As a result of this project, the following new containers were implemented as extensions of the Booch Components:

- Persistent bags (bounded, dynamic and unbounded).
- Persistent collections (unbounded) and persistent ordered collections (unbounded).
- Persistent single lists (unbounded) and persistent double lists (unbounded).
- Persistent maps (bounded, dynamic and unbounded).
- Persistent queues (bounded, dynamic and unbounded) and persistent ordered queues (unbounded).
- Persistent rings (unbounded − synchronised and guarded).
- Persistent sets (bounded, dynamic and unbounded).
- Persistent stacks (bounded, dynamic and unbounded).
- External queues.
- External stacks (fast and small).

By implementing these additional containers, many (but not all) of the original project objectives were achieved:

1. All new containers were implemented using advanced object oriented techniques in Ada 95. Performing this work therefore extended my knowledge of these techniques.
2. Twenty−two variants of eleven different persistent container types were implemented over the course of the project representing persistent versions of the vast majority of the existing volatile containers. A discussion of those that have been omitted is included below (see "Problems encountered", page 50).
3. As discussed above, implementation of external containers was given a lower priority that the implementation of persistent containers. Despite this, three variants of two different external container types were implemented which, as well as being useful in their own right, may also provide a useful framework for anyone wishing to code more external containers.
4. Also mentioned above, the final requirement to implement more volatile containers was dropped after preliminary investigation.

Additionally, it should be noted that all of the extensions to the Components were achieved without making any modifications to existing files of the Booch Components. This was identified as a desirable feature in the "Design" (page 10).

7.2 Evaluation methods used

The method of evaluation chosen for the project was to actually make use of the newly implemented containers in a "real" application. Obviously, coding a complicated fully featured application could easily form the basis of a full project in its own. Only a relatively basic application was therefore created but it was extensive enough to give an idea of the usability and reliability of the components.

The application coded to evaluate the components was a command line spell−checking program as discussed in "Spell checker" (page 45). No reliability problems were encountered during development of the application and, since the style of the new containers was consistent with the existing volatile components, using the new components was straightforward.

7.3 Overall

I therefore consider this project highly worthwhile and generally successful. Persistent versions of most of the volatile containers were implemented suitable for immediate use by anyone needing persistent ADTs. A small number of external containers were also coded for use either directly or, should an external version of any of the other containers be required, as a framework for coding additional external ADTs. I found the reliability and usability of the components in a "real−world" situation satisfactory although, obviously, some "black−box" testing using programmers familiar only with the existing volatile Booch Components would be useful. In addition to providing a useful extension to the Booch Components, the project has also achieved another important task − increasing my understanding of a number of important programming concepts including, but not limited to, generics, object−oriented techniques, iterators, storage pools, memory mapping and streams.

During the progress of this project an email was sent by Robert Leif to the Team Ada mailing list indicating that "the one key component that is unavailable in Ada is a persistent object" (see "Robert Leif request", page 85). This message served as clear confirmation that the key requirement identified for the project was indeed worthwhile.

8 Conclusion

A brief reiteration of the work done including a description of the problems encountered and acknowledgement of limitations of the project along with recommendations as to how the project could be taken further in order to eliminate any such limitations.

8.1 Reiteration of work done

Persistent versions of most of the volatile containers of the Booch Components were implemented as were a small number of external versions. This was achieved without making any changes to the existing files of the Components using object−oriented features of Ada 95.

A number of problems were encountered during the project preventing persistent versions of trees and graphs being implemented in the time allowed and introducing some limitations of the project. These are discussed below along with suggestions relating to extending the project.

8.2 Problems encountered

8.2.1 GNAT bug

Code development for this project took place at university using version 3.11 of the GNAT Ada 95 compiler and at home using version 3.12 of the compiler. No compiler problems were encountered using the later version but a problem was encountered with the earlier system. The university system would not compile the packages supporting persistent hash tables claiming that the package "Tables" was not visible despite the fact that child library units should be able to access the private items of the parent package (Barnes 1996, pp. 239−240). Believing this to be a compiler bug I reported the problem to my project supervisor and the University’s system administrator who agreed to upgrade the compiler to the later version over the Easter vacation[2].

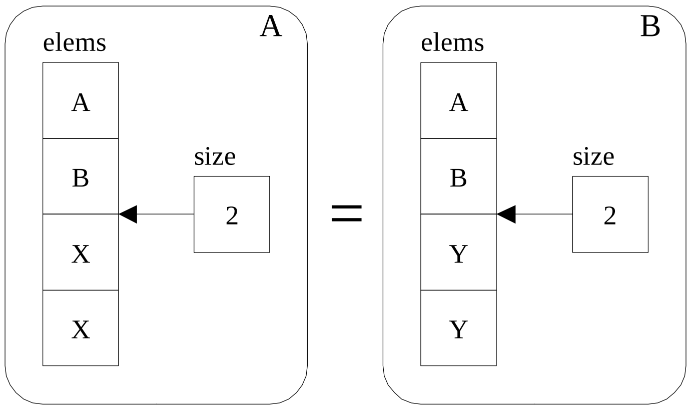
8.2.2 Booch Components bug

My test program for persistent queues indicated an error in the implementation of "Persistent_Bounded_Queue" (see "Appendix 4 − Test program output", page 82). Further investigation of this problem revealed that the error was due to a bug in the support package for bounded nodes − part of the supplied Booch Components. The problem appeared to be that no "=" function was provided to override the default comparison behaviour for the type

2 Installation of GNAT version 3.12 on the university’s systems rectified the problem.

"Bnd_Node" even though the implementation requires it. A short program, "bug_demo.adb" (see "Booch Components bug demonstration", page 81) was written to demonstrate this error and the program was sent to Simon Wright − maintainer of the Ada 95 Booch Components[3].

Figure 4 − Bug in "Bnd_Nodes" implementation.



Only un−referenced components of these two representations of bounded nodes differ but the default equality test finds the representations to be unequal because it compares all the components.

8.2.3 Graphs

Two main problems were encountered while attempting to implement persistent versions of graphs:

1. The graph data structure does not lend itself well to sequential representation as required in a persistent disk file. Nodes of a graph can be linked in a highly complex manner.
2. The volatile graph container is not derived from the abstract tagged type "Container" in the way that most of the other containers are. Consequently, much of the support code designed for children of "Container" would be unsuitable for graphs.

While attempting to solve the first problem listed above, consideration was given to the idea of using memory mapped files to implement nodes in persistent storage pools. This solution was not successful as discussed in "Memory mapping" (page 26).

8.2.4 Trees

Problems were also encountered while attempting to implement persistent versions of trees:

3 At this time no response has been received from Simon Wright regarding the reporting of the bug in the Booch Components.

1. Like graphs, the volatile trees were not derived from the abstract parent "Container" and hence much of the existing support code would have been unsuitable.
2. Although defined as a tagged types (extensions of "Ada.Finalization.Controlled") the public declaration of trees hides the tagged implementation. Persistent trees could not therefore be publicly made children of the existing volatile trees without editing the existing code. This appears to be a defect of the existing Ada 95 Booch Components that could easily be rectified. For the reasons discussed in "Design" (page 10), it was decided that no existing files were to be modified as a result of the project and hence this change has been left to the official maintainer.

8.3 Limitations

A number of limitations of the non−volatile containers have been acknowledged throughout this report. Although quite an extensive list, it should be noted than many of limitations could easily be rectified by making simple changes to the existing files comprising the Booch Components − a task that has been left for the official maintainer of the Components. Here is a summary and justification of each identified limitation:

- Access discriminants referencing a "Filename" type were chosen as the method of associating external files with non−volatile containers. The justification for this decision is given in "Persistent data" (page 14) but it has some disadvantages for example, it does not prevent two separate containers from referencing the same external file concurrently.
- Sharing of data items between separate lists is still supported but the process of saving to and restoring from a persistent file causes the shared elements to be duplicated. The reason for this is discussed in "Lists" (page 35) and was justified by claiming that a solution to this problem would greatly complicate the implementation.
- Care is required from the user to ensure that client programs call the correct overloaded version of certain routines when using external containers (in particular iteration routines and procedures for processing the first items). This is explained in "Iterators", (page 36) and "Process_Front & Process_Top" (page 38) and is imposed by the way in which the existing Booch Components have been implemented which assumes an access value can always be supplied to reference a data element of a container.
- New exceptions, introduced for the non−volatile containers, have not been declared with the existing exceptions in the package "BC". This was done to prevent modification of existing files.

- Duplicate code exists in the packages "BC.Support.Bounded.Persistent", "BC.Support.Dynamic.Persistent" and "BC.Support.Unbounded.Persistent". This was required because "Bnd_Node", "Dyn_Node" and "Unb_Node" were not implemented as tagged types derived from a common parent and hence separate routines were required to handle the saving and loading of each of them to and from disk.
- The project has been locked to version 20000103 of the Booch Components despite the existence of a later version. The reason for this is discussed in "20000219 problem" (page 42) but could be easily remedied with a simple change to the existing package implementing support for hash tables.
- Additions to the existing code that would have logically been placed in existing files were instead placed in child packages given the extension "MSB" (e.g. "BC.Containers.Bags.MSB"). This was necessary to avoid editing existing files and could easily be rectified by the official maintainer of the Booch Components by moving the additions from the "MSB" files to the appropriate file.

8.4 Taking the project further

The time constraints of the project were apparent and have limited the scope of the project. There was plenty of scope remaining to extend the project had more time been available, as discussed below:

1. The limitations listed above could have been investigated further in an attempt to determine more satisfactory solutions.
2. More persistent containers could have been implemented, i.e.
    - Persistent graphs − directed (unbounded) and undirected (unbounded).
    - Persistent trees − AVL (unbounded), Binary (unbounded) and Multiway (unbounded).

    An attempt was made at coding these persistent containers but problems were encountered (see "Graphs", page 51 and "Trees" page 51) that could not be solved in the time constraints of the project. The work done whilst attempting implementation of persistent binary trees has been included on the diskette with this project in case anyone attempts to overcome the problems encountered. The code (located in the "extend" directory of the diskette) may prove useful as it includes code that saves and restores the nodes of a binary tree from memory to and from a sequential file.
3. More external containers could have been implemented.

From the start of the project it was known that there would not be enough time to implement an external version of every container. It was therefore decided to implement just a small number in order to provide a framework for others however, if time had permitted, it would obviously have been beneficial to implement external versions of all the containers.

4. More new containers could have been implemented (in volatile, persistent and external format), e.g.
    - Dequeues (Harrison 1989, pp. 73−75) (bounded, dynamic and unbounded).
    - Priority queues (Harrison 1989, pp. 70−73) (bounded, dynamic and unbounded).

Given my interest in continuing Ada 95 development after completing my studies and the potential usefulness of the non−volatile containers to other Ada 95 developers, I intend to continue work on the non−volatile Booch Components after submission of this project. The code produced so far will be sent to Simon Wright (official maintainer of the Ada 95 Booch Components) along with details explaining the problems and limitations encountered as a result of the implementation of the existing Booch Components. Hopefully Simon will accept the merits of the work done and therefore make the changes required to the existing code in order to overcome the existing restrictions. This could lead to future co−operation between Simon and myself allowing the project to be extended in the ways described above.

At the time of submitting this report Simon had not been contacted about such issues because of the potential delays that would have been introduced by such communications (for example I have yet to receive a reply from my bug submission). An attempt at such communications will therefore be made after the project has been submitted when scheduling will be less critical.

9 References

A list of works referenced throughout the report.

- "Ada 95 Rationale" (1995), Intermetrics, Inc., 733 Concord Ave., Cambridge, Massachusetts 02138.
- "Ada Reference Manual" (1995) (version 6.0), International Standard ISO/IEC 8652:1995(E), Intermetrics, Inc., 733 Concord Ave., Cambridge, Massachusetts 02138.
- Baker, T. P. (1999), "FLORIST", Department of Computer Science, 207A Love Building, PO Box 4530, Florida State University, Tallahassee, FL 32306−4530, U.S.A. (http://www.cs.fsu.edu/~baker/florist.html; email: baker@cs.fsu.edu).
- Barbey, S., Kempe, M. & Strohmeier, A. (1993), "Object−Oriented Programming with Ada 9X" (http://www.adahome.com/9X/OOP−Ada9X.html; email: kempe@di.epfl.ch).
- Barnes, J. (1996), *Programming in Ada 95*, Wokingham: Addison−Wesley.
- Card, M. P. (1997), "Why Ada Is the Right Choice for Object Databases", Lockheed Martin Corp., Lockheed Martin Ocean, Radar, and Sensor Systems Division, Electronics Park, Building 7, Room C67, Syracuse, NY 13221 (http://www.stsc.hill.af.mil/crosstalk/1997/jun/databases.asp; email: card@syr.lmco.com)
- Carter, J. R. (2000), "PragmAda Software Engineering", DBA PragmAda Software Engineering, Concho, AZ, USA (http://www.adapower.com/pragmada; email: jrcarter@acm.org).
- "Classwide" (2000), Software Arts & Sciences, Box 891591 Houston, Texas, USA 77289−1591 (http://www.classwide.com; email: info@classwide.com)
- Gallmeister, B. (1995), *POSIX.4−−programming for the real world*, Sebastopol: O'Reilly.
- Hall, B. (1997), "Memory Mapped Files" (http://www.ecst.csuchico.edu/~beej/guide/ipc/mmap.html; email: beej@ecst.csuchico.edu).
- Harrison, R. (1989), *Abstract Data Types in Modula−2*, Chichester: Wiley.
- "Methodologists−−Rational University: Education & Training, Software Development" (2000), Rational Software Corporation (http://www.rational.com/university/rubios.html).
- Smith, M. A. (1996), *Object−Oriented Software in Ada 95*, London: International Thomson Computer Press.
- Wright, S. (1999), "The Ada95 Booch Components" (http://www.pogner.demon.co.uk/components/bc/; email: simon@pogner.demon.co.uk).

10 Bibliography

A list of works consulted but not explicitly referenced in the report.

- "Ada 95 Quality and Style Guide" (1995), Software Productivity Consortium, Herndon, Virginia.
- "Debugging with GDB" (1998), Free Software Foundation, Inc.
- "GNAT Reference Manual" (1999), Ada Core Technologies.
- "GNAT User’s Guide" (1999), Ada Core Technologies, Inc.
- Niemann, T. (undated), "Sorting and Searching Algorithms: A Casebook" (http://members.xoom.com/thomasn/s_man.htm; email: niemannt@yahoo.com).
- Rolfe, T. (1995), "Animated Sorting Algorithms (V00461)", DECUS U.S., (http://www.decus.org/libcatalog/document_html/v00461_1.html; email: ROLFE@SDNET.BITNET).

11 Appendix 1 − User guide

User instructions (how to install and use the components).

11.1 Installation

11.1.1 Overview

Like the original Booch Components, the new components are supplied in a gzipped tar file. This compressed file contains the source files of the components and, hence, these files must be extracted and compiled before use. It is also necessary to identify the location of the files to the Ada compiler e.g. by setting environmental variables.

11.1.2 Versions

Due to small differences between each version of the Booch Components, the new components are matched to a particular version of the existing components. For successful operation it is essential that matching versions of the existing Booch Components and the new components are used. The version number of the Booch Components can be determined from the release date which forms part of the name of the distributed file (e.g. "bc−20000103.tgz"). The distributed file of the new persistent components has been named so as to include the same version number as the original components to which it is matched (e.g. "bc−persistent−20000103.tgz").

Throughout the remainder of these instructions, "*ccyymmdd*" is used to indicate the version number. Each occurrence of this code should therefore be replaced with the appropriate version number.

11.1.3 Method

The distribution format of the new components is identical to that of the existing components and hence the same installation method can be used.

Due to the cross−platform nature of the components and the differing configurations of various systems, it is impossible to give instructions that can be guaranteed to work correctly for all systems. The following instructions, however, provide a guide for installing the components on a UNIX type system using the GNU Ada Translator (GNAT). Installation on other systems will depend on the tools that are available (e.g. for extracting files from gzipped tar files) and the particular Ada compiler being used.

N.B. These instructions assume that the existing Booch Components are already present on the target system. If this is not the case then they should be installed using the method described in the documentation accompanying them.

1. The gzipped tar distribution file (i.e. "bc−persistent−*ccyymmdd*.tgz" in the "install" directory of the diskette) should be decompressed and the Ada source files extracted from it. This may be achieved with commands such as:
   ```
   gunzip bc−ccyymmdd.tgz
   tar −xf bc−ccyymmdd.tar
   ```
   This should create a directory named "bc−persistent−*ccyymmdd*" containing the Ada source files that form the new components.
2. The source files should be compiled to produce the relocatable library object code. This may be achieved by typing the following into a tcsh shell:
   ```
   cd bc−ccyymmdd
   foreach FILE ( *.ad? )
      gnatmake ${FILE}
   end
   ```
   This should result in a number of source and object code files existing in the "bc−*ccyymmdd*" directory.
3. The extracted and generated files should then be moved to the required target location(s). This will depend on the configuration of the individual system. For example, if the components are to be installed for use by a single user then they may be located within a sub−directory of that user's home directory whereas if they are to be used by many users then they should be located in a globally accessible directory.
4. Once the files have been moved to their required location, the Ada compiler must be made aware of their location. If the files have been placed in directories that already contain files used by the compiler then no further action need be taken. If this is not the case then (for GNAT) two environmental variables need to be set as follows:
   "ADA_INCLUDE_PATH" should reference the directory / directories in which the source files (*.ads and *.adb) have been located.
   "ADA_OBJECTS_PATH" should reference the directory / directories in which the object files (*.ali and *.o) have been located.
   The method for setting these variables again depends on the system being used. Using the tcsh shell, an entry in a start−up file (e.g. "/etc/csh.cshrc" or "~/.tcshrc") of the form:
   ```
   setenv ADA_INCLUDE_PATH '${ADA_INCLUDE_PATH}:xxx:yyy'
   ```
   where *xxx* and *yyy* are the directory references, appends the new directories to the compiler's search path.

11.2 Usage

11.2.1 Overview

Installation of the new persistent components makes no changes to the existing volatile containers so there is no need to make any changes to any existing code that references the Booch Components or to make any changes to the way that the Booch Components are used for implementing volatile containers.

The implementation of the non−volatile containers has, as far as possible, being kept consistent with that of the volatile containers. Use of the new components should therefore be relatively straightforward, however, some differences have been necessary and these are described below. It should also be noted that the distribution includes the source code for test programs for each of the newly implemented containers. These test files, like those supplied with the original Booch Components, are relatively simple and hence provide a good example of how to use the new components.

11.2.2 Details

11.2.2.1 Package organisation

All the packages making up the abstract data types implementing the new containers are descendants of the package "BC" − from the original Booch Components. Any package that is not a child of "BC" is therefore not part of the implementation of the components and probably forms part of a test program.

11.2.2.2 Container declaration

When a persistent or external container is declared then the filename to be associated with the container must be supplied. This is achieved through the use of an access discriminant that references a "Filename" type. Filenames are implemented as a private ADT in the package "BC.Support.File" and can be created from strings using the "Create" function.

The following code provides an example of how a container implementing a persistent bounded queue could be declared. The name of the container is PBQ and the name of the associated file is "pbq.que":

```
PBQ : Persistent_Bounded_Queue(NEW Filename’(Create("pbq.que")));
```

11.2.2.3 Duplicate routines

In order to extend the functionality of certain aspects of the containers, in particular iteration, it was necessary to re−code some of the existing routines. This duplication was performed in such a manner that the new implementations of the routines in the external container packages would work with both the new external containers and the existing volatile containers. It is therefore necessary to ensure that the new versions of these duplicated routines are used in place of the old ones. This can be achieved by explicitly specifying each call or, more conveniently, by the use of a RENAMES clause to give priority to the new version, such as:

```
GENERIC PROCEDURE Modify RENAMES E.Modify;
GENERIC PROCEDURE Access_Current_Item RENAMES E.Access_Current_Item;
```

Where "E" is an instantiation of a generic external container package such as "BC.Containers.Queues.External".

11.2.2.4 Container use

The use of the non−volatile containers during program execution is almost identical to the use of the volatile containers, the exceptions being described below. The functionality for reading data items from and writing data items to the external files is handled automatically when objects are created and destroyed so no additional effort is required to handle this.

- **Data copying**: Assignment of the form
  ```
  A := B;
  ```
  cannot be used to copy the data from B into A if A and B are persistent or external containers. Instead the "Copy" routine should be used (as it is when copying data between different forms of container − e.g. from a bounded stack to an unbounded stack), i.e.
  ```
  Copy(B, A);
  ```
- **Supplementary routines**: Routines that would have logically have been implemented as additions to the existing files, have instead been implemented in additional files so as to avoid making changes to the existing components that could have introduced incompatibilities with existing code. The new packages have been coded as child packages of the existing package and given the name MSB. For example, the package "BC.Containers.Bags.MSB" implements a copy routine that logically would have been implemented in "BC.Containers.Bags". Packages ending in "MSB" must therefore be included in "WITH" and "USE" statements as required.

11.3 Tests

In order to test the installation of the components the individual container tests, additional test programs and / or spell checker demonstration program can be compiled and executed. Since the source of these test programs has been included it is also possible to examine the programs to see how to use the containers.

11.3.1 Individual container tests

An individual test program has been included for each external and persistent container. The source code for these test programs is included along with the source for the components and can be compiled in the same manner.

Once compiled, the test programs can be executed in order to test the implementation of the containers. Each test program produces text on standard output as it executes. There are two types of test performed by the test programs:

1. Persistence and primitive tests. These tests report any errors detected to standard output highlighted with the characters "**". Successful execution is therefore indicated by completion without any errors being reported.
2. Iteration tests. These tests give a messages indicating the expected result followed by the result of the iteration test. For successful tests the output produced therefore matches that described in the preceding message.

11.3.2 Additional tests

The additional test programs ("Are_Equal_Queue_Test" and "Are_Equal_Stack_Test") work in a similar manner to the individual container tests. Execution to completion without the reporting of any errors indicates that no problems were discovered.

The "Exception_Test" program should also execute to completion. It should report each of the exceptions raised and caught without terminating prematurely.

11.3.3 Spell checker

A mini spell checker application program ("Spell_Check") is also included with the source code. This program is designed to check the spelling of words in text files and is operated from the command line as follows:

11.3.3.1 Adding words to the dictionary

New words are added to the persistent dictionary by specifying the "l" (load) command line argument. When executed with this option, the program accepts words (separated by white space and / or non−letter characters) on standard input and adds them to the dictionary. Messages indicating whether each word has been stored or already exists in the dictionary are sent to standard output and error messages (e.g. if a word is too long to be stored) are sent to standard error.

As an example, if the text file "/usr/dict/words" contains a list of words to be added to the dictionary then the following command could be issued:

```
./spell_check l < /usr/dict/words
```

11.3.3.2 Listing words in the dictionary

The words stored in the dictionary can be viewed by sending them to standard output using the "s" (show) option. For example, to page through a sorted list of the words in the dictionary, the following command could be used:

```
./spell_check s | sort | less
```

11.3.3.3 Removing words from the dictionary

Words can be removed from the dictionary using the "r" (remove) option. The words to be removed should be supplied on standard input. A confirmation message will be sent to standard output or, if the word to be removed was not found in the dictionary, an error message will be sent to standard error. For example, to interactively remove words from the dictionary the following command could be issued:

```
./spell_check r
```

The user would then supply each word to be removed followed by a carriage return. The end of file character (control−d) would be supplied to finish.

11.3.3.4 Checking documents

Text documents can be spell checked with the program by using the "c" (check) option and supplying the name of the file to be checked. Any unrecognised words will be reported on standard error along with a menu of options allowing the user to interactively correct them using standard input. The corrected text will be supplied on standard output. For example, to spell check the file unchecked.txt in order to produce a correctly spelt document checked.txt the following command could be used:

```
./spell_check c unchecked.txt > checked.txt
```

Any unrecognised words will be displayed along with the following options:

- "L" (learn). This option will accept the word and insert it into the persistent dictionary so that it will be recognised in the future.
- "I" (ignore). This option will accept the word but make no record of it so any further occurrences will also be prompted for.
- "A" (ignore all). This option will accept the word and any further occurrences of it in the current document. The word will not be stored in the persistent dictionary so any occurrences in other documents will continue to be prompted for.
- "C" (change). This option will allow the user to supply a replacement for the misspelt word.

12 Appendix 2 − Structure of code

A list of source files.

The code created as a result of this project falls into a number of categories as follows:

12.1 Container implementation

12.1.1 Support packages

A number of support packages were coded to provide support routines for various aspects of persistence (see "Support packages", page 40).

- bc−support−file.ads
- bc−support−file.adb
- bc−support−bounded−persistent.ads
- bc−support−bounded−persistent.adb
- bc−support−dynamic−persistent.ads
- bc−support−dynamic−persistent.adb
- bc−support−unbounded−persistent.ads
- bc−support−unbounded−persistent.adb
- bc−support−hash_tables−persistent.ads
- bc−support−hash_tables−persistent.adb
- bc−support−nodes−persistent.ads
- bc−support−nodes−persistent.adb

12.1.2 Persistent containers

Packages exist to implement the abstract parent class "Persistent_Container" and each of the concrete external container types derived from it:

12.1.2.1 Persistent_Container

- bc−containers−persistent.ads
- bc−containers−persistent.adb

12.1.2.2 Persistent bags

- bc−containers−bags−msb.ads
- bc−containers−bags−msb.adb
- bc−containers−bags−bounded−persistent.ads
- bc−containers−bags−bounded−persistent.adb
- bc−containers−bags−dynamic−persistent.ads
- bc−containers−bags−dynamic−persistent.adb

- bc−containers−bags−unbounded−persistent.ads
- bc−containers−bags−unbounded−persistent.adb

12.1.2.3 Persistent collections

- bc−containers−collections−unbounded−persistent.ads
- bc−containers−collections−unbounded−persistent.adb

12.1.2.4 Persistent lists

- bc−containers−lists−single−msb.ads
- bc−containers−lists−single−msb.adb
- bc−containers−lists−single−persistent.ads
- bc−containers−lists−single−persistent.adb
- bc−containers−lists−double−msb.ads
- bc−containers−lists−double−msb.adb
- bc−containers−lists−double−persistent.ads
- bc−containers−lists−double−persistent.adb

12.1.2.5 Persistent maps

- bc−containers−maps−msb.ads
- bc−containers−maps−msb.adb
- bc−containers−maps−bounded−persistent.ads
- bc−containers−maps−bounded−persistent.adb
- bc−containers−maps−dynamic−persistent.ads
- bc−containers−maps−dynamic−persistent.adb
- bc−containers−maps−unbounded−persistent.ads
- bc−containers−maps−unbounded−persistent.adb

12.1.2.6 Persistent ordered collections

- bc−containers−collections−ordered−unbounded−persistent.ads
- bc−containers−collections−ordered−unbounded−persistent.adb

12.1.2.7 Persistent ordered queues

- bc−containers−queues−ordered−unbounded−persistent.ads
- bc−containers−queues−ordered−unbounded−persistent.adb

12.1.2.8 Persistent queues

- bc−containers−queues−bounded−persistent.ads
- bc−containers−queues−bounded−persistent.adb
- bc−containers−queues−dynamic−persistent.ads

- bc−containers−queues−dynamic−persistent.adb
- bc−containers−queues−unbounded−persistent.ads
- bc−containers−queues−unbounded−persistent.adb

12.1.2.9 Persistent rings

- bc−containers−rings−unbounded−persistent.ads
- bc−containers−rings−unbounded−persistent.adb
- bc−containers−rings−unbounded−guarded−persistent.ads
- bc−containers−rings−unbounded−guarded−persistent.adb
- bc−containers−rings−unbounded−synchronized−persistent.ads
- bc−containers−rings−unbounded−synchronized−persistent.adb

12.1.2.10 Persistent sets

- bc−containers−sets−msb.ads
- bc−containers−sets−msb.adb
- bc−containers−sets−bounded−persistent.ads
- bc−containers−sets−bounded−persistent.adb
- bc−containers−sets−dynamic−persistent.ads
- bc−containers−sets−dynamic−persistent.adb
- bc−containers−sets−unbounded−persistent.ads
- bc−containers−sets−unbounded−persistent.adb

12.1.2.11 Persistent stacks

- bc−containers−stacks−bounded−persistent.ads
- bc−containers−stacks−bounded−persistent.adb
- bc−containers−stacks−dynamic−persistent.ads
- bc−containers−stacks−dynamic−persistent.adb
- bc−containers−stacks−unbounded−persistent.ads
- bc−containers−stacks−unbounded−persistent.adb

12.1.3 External containers

Packages exist to implement the abstract parent class "External_Container" and each of the concrete external container types derived from it:

12.1.3.1 External_Container

- bc−containers−external.ads
- bc−containers−external.adb

12.1.3.2 External queues

- bc−containers−queues−external.ads
- bc−containers−queues−external.adb

12.1.3.3 External stacks

- bc−containers−stacks−external.ads
- bc−containers−stacks−external.adb
- bc−containers−stacks−external−small.ads
- bc−containers−stacks−external−small.adb
- bc−containers−stacks−external−fast.ads
- bc−containers−stacks−external−fast.adb

12.2 Testing

12.2.1 Individual container tests

A test program exists for each container type implemented. Each test program also makes use of a support package that instantiates the particular containers for the test and also a general support package that provides some general test routines:

12.2.1.1 General test routines

- test_support.ads
- test_support.adb

12.2.1.2 Persistent bags

- persistent_bag_test.adb
- persistent_bag_test_support.ads
- persistent_bag_test_support.adb

12.2.1.3 Persistent collections

- persistent_collection_test.adb
- persistent_collection_test_support.ads

12.2.1.4 Persistent lists

- persistent_list_test.adb
- persistent_list_test_support.ads

12.2.1.5 Persistent maps

- persistent_map_test.adb
- persistent_map_test_support.ads
- persistent_map_test_support.adb

12.2.1.6 Persistent ordered collections

- persistent_ordered_collection_test.adb
- persistent_ordered_collection_test_support.ads

12.2.1.7 Persistent ordered queues

- persistent_ordered_queue_test.adb
- persistent_ordered_queue_test_support.ads

12.2.1.8 Persistent queues

- persistent_queue_test.adb
- persistent_queue_test_support.ads

12.2.1.9 Persistent rings

- persistent_ring_test.adb
- persistent_ring_test_support.ads

12.2.1.10 Persistent sets

- persistent_set_test.adb
- persistent_set_test_support.ads
- persistent_set_test_support.adb

12.2.1.11 Persistent stacks

- persistent_stack_test.adb
- persistent_stack_test_support.ads

12.2.1.12 External queues

- external_queue_test.adb
- external_queue_test_support.ads

12.2.1.13 External stacks

- external_stack_test.adb
- external_stack_test_support.ads

12.2.2 Additional tests

The following files implement tests for the new containers that are not specific to any one particular container type:

- are_equal_queue_test.adb
- are_equal_stack_test.adb
- exception_test.adb
- mixed_queue_test_support.ads

- mixed_stack_test_support.ads

12.3 Demonstration

12.3.1 Bug demonstration

The following file was sent to Simon Wright to demonstrate the bug discovered in the existing Booch Components (see "Booch Components bug", page 50):

- bug_demo.adb

12.3.2 Application demonstration

The following files make up the spell checker application (see "Spell checker", page 45):

- spell_check.adb
- spell_check_support.ads
- spell_check_support.adb

13 Appendix 3 − Code extracts

Examples of code written.

13.1 Persistent ring streams

This code extract from the file "bc−containers−rings−unbounded−persistent.adb" shows the "Load_Data" and "Save_Data" routines of persistent rings that use streams to implement a persistent file holding the "Top" and "Mark" values of a ring as well as the data items:

```
WITH Ada.Streams.Stream_IO;
WITH BC.Support.File;
WITH BC.Support.Unbounded.Persistent;

USE Ada.Streams.Stream_IO;
USE BC.Support.File;

PACKAGE BODY BC.Containers.Rings.Unbounded.Persistent IS

   PACKAGE Persistent_Support IS NEW Unbounded_Ring_Nodes.Persistent;
   USE Persistent_Support;

   ...

   −− Retrieve data for the container from the persistent file.
   PROCEDURE Load_Data(Obj : IN OUT Persistent_Unbounded_Ring) IS
   BEGIN
      IF File_Exists(Assigned_Name(Obj)) THEN
         DECLARE
            Mixed_File : File_Type;
         BEGIN
            Open(Mixed_File, In_File, Assigned_Name(Obj));
            DECLARE
               S : Stream_Access;
            BEGIN
               S := Stream(Mixed_File);
               Natural’Read(S, Obj.Top);
               Natural’Read(S, Obj.Mark);
            END;
            Read_Data_Stream(Obj.Rep.ALL, Mixed_File);
         END;
      END IF;
   END Load_Data;

   −− Write data for the container to the persistent file.
   PROCEDURE Save_Data(Obj : IN Persistent_Unbounded_Ring) IS
   BEGIN
      IF Assigned_Name(Obj) /= "" THEN
         DECLARE
            S : Stream_Access;
            Mixed_File : File_Type;
         BEGIN
            Create(Mixed_File, Out_File, Assigned_Name(Obj));
            S := Stream(Mixed_File);
            Natural’Write(S, Obj.Top);
            Natural’Write(S, Obj.Mark);
            Close(Mixed_File);
         END;
         Append_Data_Stream(Obj.Rep.ALL, Assigned_Name(Obj));
      END IF;
   END Save_Data;

END BC.Containers.Rings.Unbounded.Persistent;
```

13.2 External exceptions

These extracts, from "bc−containers−queues−external.adb", show how the underflow exception is raised using the routines from "BC.Support.Exceptions" if an attempt is made to pop from an empty queue:

```
WITH BC.Support.Exceptions;
...

PACKAGE BODY BC.Containers.Queues.External IS

   PACKAGE BSE RENAMES BC.Support.Exceptions;
   USE BSE;

   PROCEDURE Assert IS NEW BSE.Assert("BC.Containers.Queues.External");

   ...

   −− Remove an item from the front of the container.
   PROCEDURE Pop(Obj : IN OUT External_Queue) IS
   BEGIN
      Assert(Obj.Size > 0,
             Underflow'Identity,
             "Remove",
             Empty);
      ...
   END Pop;

END BC.Containers.Queues.External;
```

13.3 Exception_Test

The following code is taken from "exception_test.adb" and demonstrates some of the exceptions of the components:

```
-- exception_test.adb
-- 2000/05/08, Mark Bond.

-- Tests exceptions raised by invalid operations on containers.

WITH Ada.Text_IO;
WITH BC;
WITH BC.Support.File;
WITH Mixed_Stack_Test_Support;

USE Ada.Text_IO;
USE BC;
USE BC.Support.File;
USE Mixed_Stack_Test_Support;

PROCEDURE Exception_Test IS

   USE Containers;
   USE SBP;
   USE SEF;
   USE Stacks;
   USE SU;
   USE SUP;

   -- Persistence not required for this test program.
   -- Therefore no filename is assigned.
   FES : Fast_External_Stack(NEW Filename’(Create("")));
   PUS : Persistent_Unbounded_Stack(NEW Filename’(Create("")));
   US : Unbounded_Stack;

BEGIN
   Put_Line("Starting...");

   -- Original volatile container.
   Clear(US);
   BEGIN
      Pop(US);
   EXCEPTION
      WHEN Range_Error =>
         Put_Line("Volatile exception caught.");
   END;

   -- Persistent container.
   -- Should behave in same manner as volatile container.
   Clear(PUS);
   BEGIN
      Pop(PUS);
   EXCEPTION
      WHEN Range_Error =>
         Put_Line("Persistent exception caught.");
   END;

   -- External container.
   -- Should raise a meaningful exception (not Constraint_Error).
   Clear(FES);
   BEGIN
      Pop(FES);
   EXCEPTION
      WHEN Underflow =>
         Put_Line("External exception caught.");
   END;

   BEGIN
      -- Assign an invalid name to a persistent container.
      -- May raise different exceptions on different systems.
      -- Cannot standardise since validity of filenames is system
      -- dependent.
      DECLARE
         PBS : Persistent_Bounded_Stack
            (NEW Filename'(Create("sam/fred.stk")));
      BEGIN
         NULL;
      END;
   EXCEPTION
      WHEN Program_Error =>
         Put_Line("Bad name exception caught.");
   END;

   Put_Line("Finishing...");

END Exception_Test;
```

13.4 BC.Support.Bounded.Persistent

The content of "bc−support−bounded−persistent.adb" follows showing the implementation of the support routines to write and read bounded nodes to and from file:

```
-- bc-support-bounded-persistent.adb
-- 2000/05/08, Mark Bond.

-- Implements support routines for persistent bounded containers.

WITH Ada.Sequential_IO;
WITH BC.Support.File;

USE BC.Support.File;

PACKAGE BODY BC.Support.Bounded.Persistent IS

   PACKAGE Item_IO IS NEW Ada.Sequential_IO(Item);
   USE Item_IO;

   -- Read saved nodes from a file.
   PROCEDURE Read_Data_File(Obj : IN OUT Bnd_Node;
                            File_Name : IN String) IS
   BEGIN
      IF File_Exists(File_Name) THEN
         DECLARE
            Data_File : File_Type;
            Buffer : Item;
         BEGIN
            Open(Data_File, In_File, File_Name);
            WHILE NOT End_Of_File(Data_File) LOOP
               Read(Data_File, Buffer);
               Append(Obj, Buffer);
            END LOOP;
            Close(Data_File);
         END;
      END IF;
   END Read_Data_File;

   -- Save nodes to a file.
   PROCEDURE Write_Data_File(Obj : IN Bnd_Node;
                             File_Name : IN String) IS
      Data_File : File_Type;
   BEGIN
      IF File_Name /= "" THEN
```

```
         Create(Data_File, Out_File, File_Name);
         FOR I IN 1 .. Length(Obj) LOOP
            Write(Data_File, Item_At(Obj, I));
         END LOOP;
         Close(Data_File);
      END IF;
   END Write_Data_File;

END BC.Support.Bounded.Persistent;
```

13.5 Spell checker hashing function

The following extract from "spell_check_support.adb" shows the implementation of the hashing function used for the dictionary:

```
FUNCTION Dict_Word_Hash(Word : Dict_Word) RETURN Positive IS
   Result : Positive := Character’Pos(To_String(Word)(1));
BEGIN
   FOR I IN 2 .. Length(Word) LOOP
      Result := Result + Character’Pos(To_String(Word)(I));
   END LOOP;
   RETURN Result;
END Dict_Word_Hash;
```

13.6 Bounded persistent queue equality test

This code extract from "bc−containers−queues−bounded−persistent.adb" demonstrated how the equality tests of persistent containers were implemented using dispatching calls to their parent’s equality test:

```
FUNCTION "="(Left : IN Persistent_Bounded_Queue;
             Right : IN Persistent_Bounded_Queue) RETURN Boolean IS
BEGIN
   RETURN Bounded_Queue(Left) = Bounded_Queue(Right);
END "=";
```

13.7 Create function for persistent dynamic containers

The following code extract from "bc−containers−queues−dynamic−persistent.adb" shows the small amount of code duplication required to implement "Create" functions for dynamic variants of persistent containers:

```
FUNCTION Create(File_Name : Filename;
                Size : Positive) RETURN Persistent_Dynamic_Queue IS
   Result : Persistent_Dynamic_Queue(NEW Filename'(File_Name));
BEGIN
   Result.Rep := Dynamic_Queue_Nodes.Create(Size);
   RETURN Result;
END Create;

FUNCTION Create(Size : Positive) RETURN Persistent_Dynamic_Queue IS
BEGIN
   RETURN Create(File_Name => Create(""),
                 Size => Size);
END Create;
```

13.8 Small_External_Stack pop procedure

The following procedure, from "bc−containers−stacks−external−small.adb", shows how a temporary scratch file is used during the process of popping items from small external stacks:

```
PROCEDURE Pop(Obj : IN OUT Small_External_Stack) IS
   Scratch_File : File_Type;
   Buf : Item;
BEGIN
   ...
   Create(Scratch_File, Inout_File, "");
   Set_Index(File_Ptr_Of(Obj).ALL, 1);
   FOR I IN 1 .. Size(File_Ptr_Of(Obj).ALL) − 1 LOOP
      Read(File_Ptr_Of(Obj).ALL, Buf);
      Write(Scratch_File, Buf);
   END LOOP;
   Delete(File_Ptr_Of(Obj).ALL);
   Create(File_Ptr_Of(Obj).ALL, Inout_File, Assigned_Name(Obj));
   Set_Index(Scratch_File, 1);
   WHILE NOT End_Of_File(Scratch_File) LOOP
      Read(Scratch_File, Buf);
      Write(File_Ptr_Of(Obj).ALL, Buf);
   END LOOP;
   Delete(Scratch_File);
END Pop;
```

13.9 Booch Components bug demonstration

The following program ("bug_demo.adb") was sent to Simon Wright to demonstrate the bug in the existing code of the Booch Components. When run, the program outputs the message "NOT EQUAL 2" indicating failure of the second equality test:

```
WITH Ada.Text_IO;
WITH Stack_Test_Support;

USE Ada.Text_IO;
USE Stack_Test_Support;

PROCEDURE Bug_Demo IS

   USE Containers;
   USE Stacks;
   USE SB;

   BS1 : Bounded_Stack;
   BS2 : Bounded_Stack;

BEGIN
   Put_Line("Starting...");

   −− Stacks are equal and this test for non−equality correctly fails.
   IF BS1 /= BS2 THEN
      Put_Line("NOT EQUAL 1");
   END IF;

   Push(BS1, ’X’);
   Pop(BS1);

   −− Stacks are still equal but this test for non−equality succeeds
   −− because unreferenced data in the bounded nodes is different.
   IF BS1 /= BS2 THEN
      Put_Line("NOT EQUAL 2");
   END IF;

   Put_Line("Finishing...");
END Bug_Demo;
```

14 Appendix 4 − Test program output

Outcome of test programs.

The following text shows the full output produced from a successful run of the test program for external queues ("External_Queue_Test"):

```
Starting External Queue Tests:
Persistence...
Persistent data found − not first run?
Primitives...
Passive Iterators:
Should see 2,3,4:
Item: 2
Item: 3
Item: 4
Should see 2,3,4 with X:
Item: 2 − Char: X
Item: 3 − Char: X
Item: 4 − Char: X
Should see list followed by maximum:
Item: 2
Item: 3
Item: 4
Maximum character = 4
Should see count of items in list:
Count = 3
Should see 5,6,7:
Item: 5
Item: 6
Item: 7
Active Iterators:
Should see 6,7,8:
Item: 6
Item: 7
Item: 8
Iterator Deletion...
Saving data to test for persistence during re−run...
Completed Persistent Queue tests.
```

The following text shows extracts from an unsuccessful run of a test program. The error (marked with "**") indicates a test failure. In this instance the failure is due to the bug in the implementation of "Bnd_Node" (see "Booch Components bug", page 50) rather than an error in the implementation of the persistent queue being tested.

```
Starting Persistent Queue Tests:
...
Primitives:
Bounded...
** T25: Queues are not equal
Dynamic...
Unbounded...
...
Completed Persistent Queue tests.
```

15 Appendix 5 − Communications

Communications made during the course of the project.

15.1 Simon Wright

The following email, sent from Simon Wright (maintainer of the Ada 95 Booch Components) to myself during the preliminary research phase of the project, gives some justification for my decision to concentrate on implementing persistence for the components:

Hi Mark,

> I am a final year student at Aston University and am just about to start
> work on a final year project. For this project I am considering working
> on some expansions for the Ada 95 Booch Components. In particular, some
> of the ideas I have been considering include:
>
> 1) Coding components which you intend to support but which are not
> presently available (for example; dequeues and guarded stacks).

The reservation I have is that the BCs are, as literally as possible, translations of a set of C++ components, so (a) you would need access to the C++ (which I'm sure Rational would allow, but I haven't asked them) and (b) I'm not sure what value such an approach would have to you from the final year project p.o.v.

> 2) Coding additional components for which you don't mention any
> intention for future support (for example; priority queues and hash
> tables).

The BCs do have ordered queues (which I have just done) −− are these the same as priority queues?!? −− and the support for Maps, Sets, Bags uses hash tables ..

> 3) Coding some non−volatile data structures based on existing components
> (e.g. a disk file that represents a queue). I can see two possible
> approaches to this:
> a) Loading the complete data structure into memory on initialization and
> saving the complete structure back to disk on finalization.
> b) Maintaining the structure on disk and loading elements from disk only

> on demand.

This sounds like a good idea. There is support for persistence in the C++ BCs
but I haven't looked at it and doubt very much whether I will.

> Regarding the above points, I would appreciate your answers to the
> following queries and would also be happy to hear any other suggestions
> or comments you might have regarding the proposed project.
>
> 1) What do you mean by the component you call a "sequence"?

Umm, I don't know −− no such thing the the C++ BCs! must be a case of
documentation enthusiasms on the part of my predecessor, I guess.

> 2) Is an update to the Booch Components likely in the near future that
> would render much of the work I plan to do redundant?

Minor updates (bugs; sorted queues; AVL trees made non−limited). Still in
the mill: the ordering rules for insertion of multiple elements with the same
key in ordered collections and queues are stupid.

> 3) Would you offer any recommendations as to which components I would
> be best to work on? Are there any components that are close to
> implementation and would therefore not be worth my time? Are there any
> other types of data structures that you consider need implementing that
> I could perhaps work on?

On the whole I think persistence would be best if that seems good to you.
There's no way we would be tripping over each other then.

Best wishes,

−−

Simon Wright

15.2 Robert Leif request

The following email, sent from Robert Leif to the Team−Ada mailing list, confirmed that persistence would be a worthwhile addition to the Booch Components:

From: Bob Leif

To: Simon Wright et al.

As far as I could discern, the one key component that is unavailable in Ada is a persistent object. Michael Card has demonstrated that this is feasible with his FIRM database. However, FIRM presently appears to be unavailable and the software is owned by Lockheed. My experience with Lockheed is that I have a very high regard for the corporation's technical ability, very high admiration for Michael Card's work, and very little hope that Lockheed could ever successfully market commercial software.

I deliberately did not use the term persistent_object class, because a requirement should not specify an implementation and Ada's object oriented programming repertoire includes tagged types, generics, and elegant combinations of both.

15.3 Robert Leif reply

The following email, sent from Robert Leif to myself during the course of the project shows Robert Leif's request for an inclusion of a demonstration of bounded strings with the persistent components:

From: Bob Leif

To: Mark Bond

Sounds excellent. I hope that you put a tagged type inside of a generic. That way we can use any standard Ada data type and an automated system can be built where the user can fill out a form. If you get a chance, please look at Michael P. Card's FIRM database. One very important generic type is bounded strings. You should include an example. The package should be extensible to handle complex data types like arrays; however, it would be inappropriate for you to have to do that for a final year B.Sc. project.