A lightweight class library for extended persistent object management in C++

Jens-Uwe Dolinsky, Thorsten Pawletta

University of Wismar, Department of Mechanical Engineering, Process and Environmental Engineering, Philipp-Müller-Str., P. O. Box 1210, D-23952 Wismar, Germany; E-mail: dolinsky@amundsen.mb.hs-wismar.de, pawel@bering.mb.hs-wismar.de

Abstract. When applications must keep their data structures persistent, additional effort beyond transient modeling is necessary to realize suitable persistent storage. The most convenient approach is the use of a persistent programming language (PPL), which offers internal mechanisms for storing and loading data transparently for the application programmer. An alternative to a PPL is a universal programming language that has been extended by persistence concepts for any data types.

This paper introduces an extended generic mechanism that abstracts necessary functionality for realizing persistence of any C++ object structure. First, we discuss general problems of persistent storage and the motivation for this approach. The design aspects and the reasons for special features are described in detail with the data and implementation inside. Finally, we sketch the persistent data structure and present the integration of the library in three different example programs.

Key words: Modular-hierarchical – Persistence abstraction – Persistent modelling – Data type orthogonality

1 Introduction

The lifetime of data values from their creation until their deletion is a measure of their persistence. This period can be very short, e.g. temporary results of calculations, or very long if data outlives versions of persistent support systems (see Atkinson and Morrison [1]). In this context computer data are classified as short-term data (transient main memory data) and long-term data (data independent of program lifetime, or persistent data). One of the major problems is the different semantics of short-term and long-term data, which requires different access methods. Long-term data exist outside and independent of the creating program in external files or databases. For processing within an application, the long-term data has to be converted into directly accessible transient formats and afterwards must be returned to persistent formats again. In the worst case, data has to be converted between different data models (e.g., relational ↔ object-oriented), which can prove inconvenient and causes for inconsistencies and information loss. This means that the application programmer has to consistently maintain two formats (transient ↔ persistent) of the same data, which can be a hard job.

One solution is the integration of concepts supporting data persistence into a programming language. This can make the processing of data independent of persistence. For the programmer, invisible mechanisms automatically load and save the data and convert them between persistent and transient structures [6].

In the object-oriented world, data persistence means the persistent storage of complex, polymorphic, modular-hierarchical object structures. (The linkage of the methods is not discussed here; see Dolinsky [4].) Some object-oriented programming languages (e.g., Smalltalk) already support persistence in the language standard. In contrast, C++ leaves the programmer to implement persistence mechanisms for data objects with all the consequences. In the past a lot of different solutions were developed. Because of the special requirements of a C++ based modular-hierarchical simulation run-time system for variable structure systems [8] (e.g., separate persistence of inherited data), we propose an approach that can be considered as an extended general solution for persistent object management.
2 Problem analysis

On the way to persistent objects, many different physical and logical problems have to be solved. The main task of a persistence mechanism is the transformation of main memory object structures into flat sequential byte streams. The persistent storage of a complex object structure must include the values of the individual objects and the relations (references) among them. References in the transient structure (pointers) have to be translated into object numbers because identification of the persistent objects in the stream does not work with main memory addresses. Access to persistent objects via object numbers presupposes a predefined order of the objects in the stream.

Another important problem is the storage of cyclic structures. The persistence mechanism has to traverse the entire structure with a specific strategy (e.g., depth-first order) to find all objects. In this context an endless loop must be prevented during the search.

The program-independent existence of persistent data causes a further general problem. The type bindings of persistent data in the stream cannot be protected by the type system of the programming language. To keep the semantics of the persistent data, the persistence mechanism has to prevent incorrect interpretation of the data e.g., by using redundant information in the persistent storage.

Among the special requirements of the persistence mechanism is the ability to handle substructures used by multiple objects, such as in Fig. 1. If Object 1 is to be stored, its referenced objects (Object 3) or substructures (Object 5 and its referenced Objects 6 and 7) have to be stored as well because of their logical correlation (persistence by reachability). If Object 2 is to be stored, Object 4 and the substructure Object 5, which is used in common with Object 1, have to be saved too.

The result is the existence of two independent persistent copies of the substructure Object 5. After reconstruction, Object 1 and Object 2 do not share the same substructure; both objects have their own local copy of Object 5 and its components (Fig. 2), which inflicts unacceptable information loss:

Another special requirement is the selective storage of single objects or substructures out of the context of a wrapper structure (see Pawletta et al. [8]). In the example illustrated in Fig. 3, the single Object 2 is to be stored out of (or loaded into) the context of the entire structure. Therefore the search of the persistence mechanism for not stored/loaded objects must be disabled (e.g., by parameterizing).

Another required special feature is storing/loading inherited data separately [8]. This means that an object must additionally be able to explicitly store/load components that are inherited (specified by the name of the inherited class; also see the implementation examples in Sect. 6).

The general technical problems are described in the following sections.

3 Design principle

The persistence mechanism presented here is encapsulated by a generic base class that can be inherited by each user-defined class (data type orthogonality). The major aims were portability of the entire solution, transparent mechanisms and a minimum of complexity (no global data).

The transparency of the algorithms is achieved by their total encapsulation based on the functional abstraction of the persistence mechanism.

The components of the generic class can be divided into completely encapsulated components (not invisible and therefore not adjustable by inherited classes), interfaces and persistence methods. The set of encapsulated components encompasses File Management, Object Management, Garbage Collection and Error Handling (see Fig. 4).

The File Management is responsible for creating, opening and closing files as well as reading and writing the data. Because of the encapsulation of this functionality, the persistence mechanism of the generic class and thereby implicitly of all inherited classes is easy to adapt to other read/write devices, e.g., sockets, pipes.

The task of Object Management is the administration of objects during storing/loading processes. It registers

Fig. 1. Substructure used by multiple objects

Fig. 2. Loss of commonality of substructure after reconstruction

Fig. 3. Selective storage