A Study into the Feasibility of Generic Programming for the Construction of Complex Software

Santi Caballé[1] and Fatos Xhafa[2]

[1] Open University of Catalonia, Department of Information Sciences
Av. Tibidabo 39-43, 08035 Barcelona, Spain
scaballe@uoc.edu
[2] Polytechnic University of Catalonia, Campus Nord C6
Jordi Girona 1-3, 08034 Barcelona, Spain
fatos@lsi.upc.es

Abstract. A high degree of abstraction and capacity for reuse can be obtained in software design through the use of Generic Programming (GP) concepts. Despite widespread use of GP in computing, some areas such as the construction of generic component libraries as the skeleton for complex computing systems with extensive domains have been neglected. Here we consider the design of a library of generic components based on the GP paradigm implemented with Java. Our aim is to investigate the feasibility of using GP paradigm in the construction of complex computer systems where the management of users interacting with the system and the optimisation of the system's resources is required.

1 Introduction

In traditional forms of engineering we find that productivity, quality and cost are such important factors in industrial processes that the very survival of companies depends upon them. For this reason, great efforts have been made to improve the techniques, methods and tools which are available for product development and the results have clearly been spectacular. However, in the case of software development similar progress has not been made so far even though there is no essential difference with other forms of engineering. In explaining why this is so we need to remember that the technology is more recent and that the product is highly complex, but the key is doubtless to be found in the fact that the concept of reusability has not been sufficiently exploited.

The *reuse* of previously created product parts leads to reduced costs and improved productivity and quality to such an extent that industrial processes will take a great leap forward. In all advanced forms of engineering we can observe that new products are usually developed by reusing tried and tested parts but in software engineering it seems that new products are almost always developed from scratch. To benefit from the advantages of reusability it is necessary to develop better methodologies such as Generic Programming (GP) that facilitate

this possibility. GP when applied in the context of computer software development is an innovative paradigm that attempts to make software as general as possible without losing efficiency. It achieves its goal by identifying interrelated high-level family from a common requirement set. By the application of this technique, especially in design phases, software is developed offering a high degree of abstraction which is applicable to a wide range of situations and domains.

2 Generic Programming Objectives

By applying GP to develop computer software we achieve important objectives:

Reuse. This is the main reason for applying GP ideas. The real objective, however, is to be able to reuse and extend software components widely so that it adapts to a great number of interrelated problems. This concept of reutilization is much broader than has been seen until now.

Quality. Due to the great potential for reutilization of GP, it is necessary to guarantee maximum quality. Here "quality" refers to the correctness and robustness of implementation which provides the required degree of reliability. Furthermore, GP provides innate reliability as the implementations are nothing more than skeletons, without details, and as such are simpler to construct during the codification phase and hence can be produced with minimal errors.

Efficiency. As in the case of the quality, we also have to guarantee the efficiency of components as if this not done the performance repercussions will be noted, just as with lack of quality, in all of the systems involved.

Productivity. Inherent to reutilization is the saving through not having to create software components again that already exist. Hence, there is an increase in computing production–one of the benefits that GP seeks in order to bring the production potential of computing applications as in other industries.

Automatisation. Here the aim is to automatise the processes so that general requirements with a high level of abstraction and specially designed tools can be used to produce operative programmes. GP provides the skeleton for the initial generic requirements. The resulting programme will be the base for other more specific programmes which in turn will construct others following a cascade process so giving us a more solid base permitting more automatisation.

Personalisation. As the general requirements are made more particular, so the product that is generated becomes more optimised to meet the specific needs of the client. GP intervenes largely in a first phase of this abstraction-specialisation/personalisation cycle by identifying the abstractions and supplying a first level of specialisation; this is a great challenge of the software industry.

3 Statement of the Problem

The development of applications based on GP not only has the practical advantage of improved productivity but also results in software that is more robust and reliable. Considerable research has already been done into the construction

of libraries of generic data structures based entirely on GP (Standard Template Library [1] and Data Structures and Algorithms in Java [2]) but very little has been done into the development of computer systems.

As a contribution to filling this gap, we consider the construction of a library (see API at: http://cv.uoc.edu/∼scaballe/tfc/api) of components of greatly generic use that creates the skeleton for the construction of complex systems requiring the management of the users interacting with the system and optimisation of the system's resources. The aim of this study is to investigate the feasibility of the design of generic software as the basis of complex and extensive-domain computer systems. The library is based on the GP paradigm as the aim is to encourage the greatest possible reusability of its own generic components for the development of the specific computer systems highly complex. Our first step is to identify those parts which are common to most applications of this domain and we then proceed to isolate the fundamental parts in the form of abstractions from which the basic requirements have been obtained. Once a logical division into components and subsystems has been made, we analyse and design each component separately employing OO methodology. In order to maintain intact the ideas of GP design that are found, we implement an implicit logical layer that creates a correspondence between the GP design and the OO design.

Since Java has a great predisposition to adapting and correctly transmitting a high degree of abstraction and make the software reusable, this library has been implemented in this programming language.

As we will demonstrate, the advantages that GP offers with regard to quality, efficiency, productivity and so on provides a solid basis for the construction of specific software that is faster, more efficient and highly reliable.

4 Design of the Library

The library is made up of three components which constitute the skeleton of the basic structure of whatever application is constructed using it.

The Users component. It is made of two subsystems: (a) **GenericUserManagement.** It administers the basic elements participating in the administration of the users of a computing system(see Fig. 1). The key entity *GenericUser* represents a person, group, device, etc., which is able to have multiple identities/roles. An *Identity* is unique both at user level and within the group of all the users of the system. An *Authenticator* is made up of both public and private keys and may be used to validate each identity. (b) **UserProfileManagement.** It administers user preference information obtained through the elements of the profile. The sum of all of these *ElementProfile* forms the *UserProfile*.

The Security component. It is made up of two subsystems: (a) **Authentication-Management.** It identifies the user and manages the log in and log out procedures. The generic process of authentication consists of starting a *UserSession* in which a user's identity and authenticator are validated. (b) **Authorisation-Management.** It administers the system's security policy. This information will

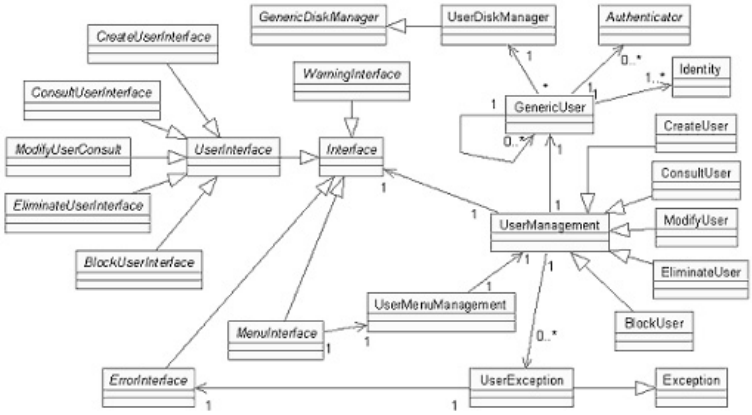


Fig. 1. Class diagram for GenericUserManagement subsystem.

essentially be obtained from users permission to limit access to system's valuable resources.

The Control component. It is made up of two subsystems: (a) **SystemMonitoringManagement.** It administers all of the data produced by the system itself as a result of the events occurring during normal use (see Fig. 2). The key entity of this subsystem is the *History* made up by the events in the day-to-day operations of the system. Each event is recorded in an *ElementLog* containing the date and time the event occurs together with a textual description. (b) **PerformanceControlManagement.** It administers the statistical information about the system (stored in the *History*). The *Statistics* entity will enable us to extract useful information about the performance and reliability of the system.

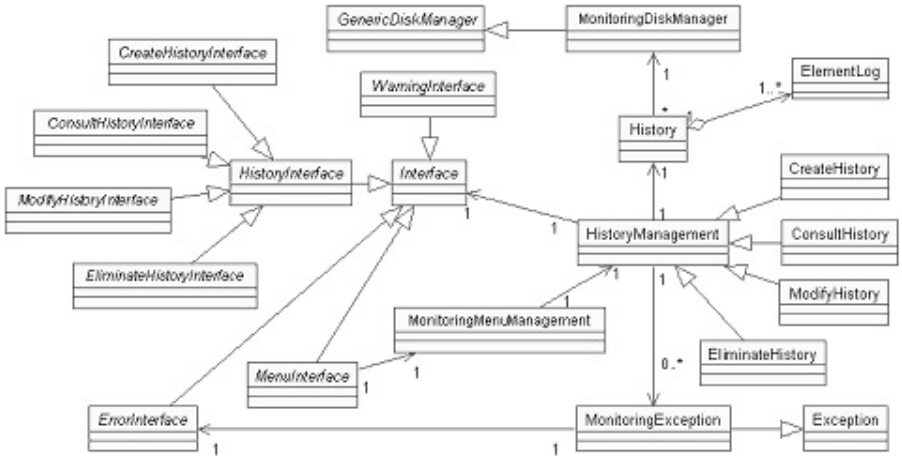


Fig. 2. Class diagram for SystemMonitoringManagement subsystem.

For each component, the *user interface* is generic and permits particularisation both in text and graphic modes. With regards to the persistence of the data, we consider the existence of an abstract database which through a *generic disk manager* will act as a bridge between the future application and its data in order to make the design independent from the persistence of the specific technology

managing the data and which will allow the treatment of both ordinary texts and the database during particularisation.

A complete hierarchy of exceptions has been created to capture and treat errors and anomalous situations that are not foreseen in the specification and which come to light whilst working with each component. These exceptions give a high degree of reliability to applications built with the components without having to depend on the error treatment of the specific platform.

5 An Application: Use of the Library in Constructing Distributed Applications for Collaborative Learning

We have gained experience in GP paradigm by using our library as the base for constructing complex applications within the domain of the library itself.

We are currently working on the development of a prototype of a collaborative workspace as a distributed and decentralised application (this kind of distributed applications are discussed in [7]) aiming to facilitate working groups as well as asynchronous groupware coordination tasks in the context of Project Based Collaborative Learning. The main objective is to create web-based shared workspaces where the users of each space are able to deposit, receive and share all types of resources permitting full synchronous and asynchronous communication without depending on the computer platform being used. The system permits the identification and the notification of recent events to each shared workspace user and the consultation of the system events history in order to extract statistics about any time period of the system's life. Furthermore, the system permits the control of user access to valuable resources by allocating roles and privileges to all system users.

Due to the high degree of interaction between users and the system (and particularly the fact that the events are widely spread) and the necessity to control and optimise the system's resources (with minimum storage), we can see that the system domain forms part of the library's domain showing that the library is clearly reusable in this area. What distinguishes our system from existing client/server environments such as BSCW [8] is the strong decentralisation of the system which has great advantages in terms of the increased hardware resources, high performance, lower cost and strong scalability which are all essential factors.

In order to decentralise the system, a P2P architecture has been designed especially for document sharing (text, audio, etc.) among the users. On the other hand, the system resources index and the system administration in general (user authentication, system's events management, etc.) is centralised in a single point as attempting to decentralise this part provoked a worsering in the system's behaviour in terms of document location, user management, system security and so on. During the construction of our application, we have checked how even in design phases our library adapts perfectly thanks to good matching of those generic entities proposed with the instantiation of those we have made here. We can see, for example, how *GenericUser* matches with whatever user type

is found in a collaborative work environment (single users, groups, etc.) and *Identity* becomes the unique identifier of the user (e-mail address) and we can also check how *ElementLog* gathers all the information needed to identify the system's events from which it is possible to notify to the rest of the users. With regards to functionality, the generic processes are also well-matched with specific ones such as *AutenticationValidation*, which once instantiated, allows users to be authenticated and access to their own workspace.

The user interface is instantiated in a graphic mode taking advantage of the genericity of the library. Regarding persistence, *GenericDiskManager* abstraction and its specialisations are available representing a bridge that keeps the logic of the application independent from its data allowing persistence in different models, which in our case will be Scalable Distributed Data Structures [6].

During the design of the decentralisation, we have checked that our library is not totally reusable with respect to persistence as it was not designed for this aim. A generic data structure interface needs to be designed which will permit data to be distributed and located within a scalable, consistent and fault-tolerant context (the latter basically through data replication). Even so, thanks to the great GP abstraction capacity, it permits one more grade of genericity to be imposed in the library so preparing it for distributed and centralised environments.

Conclusions and Ongoing Work. In this study we have shown the feasibility of generic programming for the construction of complex software within a specific but extensive domain. We would like to assess the performance of generic components when specialised in specific systems (e.g. distributed systems over the Grid) and to estimate their development cost. We will also investigate other facets of GP such as design patterns and generic exception handling and provide another version of our library for distributed and decentralised environments.

Acknowledgments. This work has been partially supported by the Spanish MCYT project TIC2002-04258-C03-03.

References

1. Musser, D., Derge, G., and Saini, A: The STL Tutorial and Reference Guide. C++ Programming with Standard Template Library. Addison-Wesley, 2002
2. Goodrich, M. and Tamassia, R.: Data Structures and Algorithms in Java. Willey 2001
3. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison Wesley, 2001
4. Bloch, J.: Effective Java Programming Language Guide. Addison-Wesley, 2001
5. Kreft, K. and Langer, A.: Combining OO Design and Generic Programming 1997. http://www.langer.camelot.de/Articles/OOPvsGP/Introduction.htm
6. Litwin, W.: SDDS: Scalable Distributed Data Structures, 2000. http://ceria.dauphine.fr/SDDS-bibliograhie.html
7. Borghoff, U.M and Schlichter, J.H.: Computer-supported cooperative work: introduction to distributed applications. Springer, 2000.
8. Bentley R., Horstmann T. and Trevor J.: The World Wide Web as Enabling Technology for CSCW: The case of BSCW. Computer-Supported Cooperative Work: Special issue on CSCW and the Web, Vol. 6. 1997 Kluwer.