Serdica
Bulgariacae mathematicae publicationes

Сердика
Българско математическо списание

For further information on
Serdica Bulgaricae Mathematicae Publicationes
and its new series Serdica Mathematical Journal
visit the website of the journal http://www.math.bas.bg/~serdica
or contact: Editorial Office
Serdica Mathematical Journal
Institute of Mathematics and Informatics
Bulgarian Academy of Sciences
Telephone: (+359-2)9792818, FAX:(+359-2)971-36-49
e-mail: serdica@math.bas.bg

SERDICA — Bulgaricae
mathematicae publicationes
19 (1993) 80-85

BUILDING A PARTIAL SOFTWARE ENVIRONMENT[†]

NELLY MANEVA

1. Introduction. Several modes of automating software production are known. Some authors consider that the tools used most often are the so called "Stand-Alone" ones. The latter enable us to solve only a single problem and they may be used in special environments. Testbeds, language-oriented editors, project-schedulers, advisers, etc. can serve as good examples in this respect.

The development of Software Engineering Environments (SEE) supporting all the basic functions during the Software Life Cycle (SLC) (e.g. SOFTING [5], NASTEC-CASE 2000 [2]) is another approach to this problem. Unfortunately, the conclusions made about the real usage of these systems are not very optimistic. Their applications depend on the necessity of observing some established requirements and standards for all the SLC phases. SEE are also avoided because of complexity, the high prices and the long period of time needed for their creation. The proposed unifying principles formulated in such a general way provide almost no useful specific guidance and are another reason for their limited use. When implementing a SEE one should consider a great variety of factors. Some of these factors are related to the organization concerning software production, e.g. the model of SLC used, the style of operation chosen. Other factors refer to the project under development, namely, its application area, size, complexity, duration and number of people involved.

Taking into consideration the facts given above, we have made an attempt to develop a feasible approach which appears to be a compromise between two extremes, namely the individual tools and the powerful SEEs.

The present paper describes the principles proposed for building **partial** software environments (PSE) (i.e. environments which provide only some of the facilities needed).

The concept of PSE is introduced in Section 2. In Section 3 the author investigates the results of implementing PSE example in the field of program analysis and

[†] This research has been partly supported by the Ministry of Education and Science under grant No I-24.

quality evaluation. This section comprises the methodology proposed and the overview of the designed automated system. A generic framework for building PSEs is given in Section 4. Finally, the author's conclusions and some ideas on further research in this area are summarized in Section 5.

2. Partial Software Environment (PSE). The existing general purpose SEE intend to cover all the activities for all the SLC phases [3]. These SEE are not flexible enough so as to meet the needs of a given software project, the style of a certain organization and the individual developer's methods.

That is why we propose to restrict the scope of SEE by building **partial** ones. We are going to stick to the idea that only **a few, interconnected and important activities in the course of some SLC phases should be automated.** We think that dealing with a smaller subset of activities related to software creation and evolution may ensure better support of these activities.

The following steps should be observed for creating PSE:

1) To select the activities;

2) To establish a systematic methodology through integrating some methods related to the basic principles of the activities chosen;

3) To design an automated system so as to support the methodology.

That system should meet the following requirements:

– to be an integrated set of several components which supplement each other and whose joint applications can support the selected activities;

– to have a modular structure enabling the easy extension and replacement of components;

– to include tools providing a few (at least two) levels whose complexity, depth and resource requirements vary.

– to use data organization and user interface independent of the activities chosen;

– to allow collecting information about its own performance and thus to enable the evaluation of its efficiency.

3. PSE Example. The proposed approach has been applied to the field of program analysis and quality evaluation. We have tried to deal with problems related to ensuring high quality software products. First, it is necessary to define what would be meant under a "quality software product" and to select those activities which would affect that quality and which might be automated.

By **software product of quality** we mean a product which meets the requirements formulated for it.

Next we shall consider the problems of program analysis and quality evaluation. Both activities are connected with examining programs. Analysis checks whether preliminary defined requirements are met. The main purpose of quality evaluation is to

determine **how** (degree, efficiency, amount of efforts expended, etc.) these requirements have been met by the implemented program.

By **analysis** we mean the program examination, aiming at checking whether the programs are in correspondence with the preliminary defined requirements.

Any deviation from the given requirements is called a **defect**.

We think that **analysis**

– aims at discovering defects and at accomplishing their removal;

– is performed by the developers (nowadays it is usually carried out by independent persons with special control functions);

– applies "bottom-up" strategies to the program units examined with respect to size, structure and complexity;

– is supported by a set of automated tools.

Also we consider that the following rules should be observed with the **quality evaluation**:

– the developers creating the components of a software product who are responsible for its quality should perform the "inner" evaluation themselves;

– only programs being a concrete implementation of the algorithms chosen at a certain design phase should be evaluated;

– evaluation should be carried out at various stages of program development and at several decomposition levels;

– both static and dynamic program characteristics should be evaluated;

– all measured quantities should be calculated automatically by tools;

– the evaluation should be completed by analyzing the results obtained by taking into account the specific conclusions made and the constructive instructions formulated.

In order to use this approach to quality evaluation, we have to create a set of appropriate metrics. These metrics should be validated (i.e. the effect of their application should be theoretically or empirically confirmed). Besides metrics should propose measurement procedures which might be entirely performed by automated tools.

A system supporting the analysis and the quality evaluation activities and ensuring the efficient and systematic use of the proposed methodology has been designed. For more details about this system see [1]. In the present paper we shall only describe briefly its components.

The **formatter** produces a source code listing, which shows the number of each statement and the levels of indentation.

The **modifier** creates variants of a certain program by modifying it.

The **configurator** forms a named collection consisting of elements which might be either of the same type or of different types.This collection has to be processed by other tools in the system.

The **static analyzer** performs simple static analysis,data flow and control flow analysis.

The **dynamic analyzer** runs a program in a controlled and systematic way so as to determine its functional correctness or incorrectness.

The **quality evaluator** is a collection of tools implementing the measurement procedures of the set of metrics chosen.

The **report writer** creates various kinds of reports and displays them.

The information to be maintained for the analysis and the quality evaluation ranges from textual source code to different results obtained after using some tools (e.g. tables, intermediate representations, error messages, etc). The **Data Base (DB)** of the system stores this information.

The **command interpreter** provides an advanced interface between the system and the users and it is an essential component of the system. It maintains the DB so as to guarantee its structural and functional integrity. Also this interpreter hides some details of implementing complex operations.

A prototype of the proposed automated system has been created under UNIX System V operating system on the ICL CLAN computer. The program system SET (Software Evaluator and Tester) examines C-programs and it is written in C itself. All the requirements formulated in Section 2 have been met by SET.

Although the main purpose of the automated system is to support program analysis and quality evaluation, it has proved to be useful for some other activities such as program development, optimization, maintenance and project management [1].

4. How to build PSE. The SET implementation has facilitated the experimental study of the various aspects of PSE which supports program analysis and quality evaluation. The use and the efficiency of this PSE have been evaluated which resulted in formulating some general principles about building a PSE.

Our efforts have been directed to developing a generic framework that might be used as a basis for creating different PSE instances. Taking into account the great diversity of selected activities chosen to be supported by a specific PSE, we have tried to clarify their common features and their differences. Thus two phases in the process of building a PSE might be distinguished:

Phase 1. Specifying a PSE. This preliminary phase involves entirely Step 1 (selecting the activities), Step 2 (establishing a systematic methodology – a set of guidelines, rules and strategies) and partly Step 3 (defining the components of the automated system). Thus independently of the use of a PSE this phase would always result in describing the automated tools (ready-made as well as ad hoc ones) and in describing the objects these tools operate on.

Phase 2. Constructing a PSE – creating an integrated system satisfying the conditions formulated in Section 2.

To fill the gap between these phases two significant architectural problems

should be solved:

– how to organize the information repository in the PSE;

– how to integrate and/or to adapt the tools needed.

We shall suggest a solution concerning the universal data organization and the respective user interface.

We assume that the **Information Repository** comprises three types of objects: basic objects, schedules and derivatives.

A **basic object** is a unit which a certain tool operates on.

Some characteristics of the basic objects (name, status, owner, date of the latest updating, version, etc.) can be described by attributes. The attribute types can be scalar, enumerated or hierarchical. The attribute numbers, types and meanings have to be defined only once when starting the development of a certain software project. The basic objects can be projected and retrieved in various ways and some of the actions can be performed automatically by using the values of the attributes.

A **schedule** is a unit which controls the application of a tool.

There are different types of schedules, corresponding to the tools chosen. A schedule describes different (in content and capacity) capabilities and, if necessary, it gives additional information related to the way of applying the tools. The user is allowed to create his own schedules, by selecting various alternatives available. Each tool is supplied with a default schedule.

The use of schedules ensures a flexible and efficient application of the tools.

A **derivative** is a unit containing the results obtained due to a tool application.

It is not necessary to save the derivatives in the system **Information repository** permanently. Having applied consecutively the various tools for achieving a certain aim and displaying the results obtained one might delete the derivatives which are not needed anymore.

The user interface we propose is command driven. The synopsis of each command is:

```
@<name> <list of arguments>
```

The commands are divided into three groups:

i) **Data control commands.**

They perform actions related to the **Information Repository** as a whole (its creation and deletion, definition of the number, meaning and type of attributes) or particular objects in the **Information Repository** (addition, deletion, retrieval, renaming, updating, etc.)

ii) **Utility commands.**

They provide information about Information repository content and about the tools available.

iii) **Tools control commands.**

The synopsis of these commands is as follows:

`@<name> <object> [<schedule>]`

`<name>` points out the tool which should be invoked, while `<schedule>` controls its use. If the second argument is omitted, then the default schedule for this tool will be used.

5. Conclusions. The present paper aims at introducing the concept of PSE covering only a few activities during some SLC phases. An instance of PSE confirming some advantages of the proposed solution for software production automation has been presented. The phases and the main components of the generic framework for building PSE (data organization and user interface) have been outlined. Some interesting aspects of further research in this area arise. More sophisticated user interface might be developed. A knowledge based approach to building PSE would be very helpful in the Specification phase. Some encouraging results in this direction have impressed us, e.g. the expert system TABA [4] assisting a software engineer in choosing the components of the most suitable SEE for the particular application at hand.

REFERENCES

[1] N. Maneva, Architecture and possibilities of a system for program analysis and quality evaluation, Proc. of conf. on Automated Teaching and Information Systems, Varna, 1986, 168-176.

[2] H. J. Manley, Computer Aided SE (CASE) – Foundation for Software Factories, Proc. CompCon 84 – The Small Computer (R)evolution, Arlington, 1984.

[3] Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, (P. Henderson, ed.), Palo Alto, California, 1986.

[4] A. da Rocha, J. M. de Sonza et al., TABA – A Heuristic Workstation for Software Development, Proc. of Int. Conference on Computer Science and SE, 1990, 126-129.

[5] H. Sneed et al., Automated software quality assurance, *IEEE TSE* **SE-11** (1985), 909-917

Software Engineering Department
Institute of Mathematics
Acad. G. Bonchev Str., Bl. 8
1113 Sofia
BULGARIA

Received 26.11.1992
Revised 28.12.1992