1. Introduction

Re-engineering is the examination, analysis and alteration of an existing software system to reconstitute it in a new form, and the subsequent implementation of the new form (Rosenberg, 1996). The purpose of re-engineering is to understand specification, design and implementation of some existing software and to re-develop this software in order to achieve a higher degree of functionality, security, reliability, etc., in other words to enhance the software. There are four general re-engineering objectives (Rosenberg, 1996):

- preparation for functional enhancement – to specify the characteristics of the existing system that can be compared with specifications of the characteristics of the desired system,
- improve maintainability - to re-design the system with more appropriately functional modules and explicit interfaces,
- migration – to migrate to a newer hardware platform, operating system, or language,
- improve reliability - the reliability of the software steadily decreases to the point of unacceptable

2. Phases of software development

The main phases of software development, according to the level of abstraction are (Rosenberg, 1996):

- Conceptual abstraction – functional characteristics are described only in general terms.
- Requirement abstraction – functional characteristics are described in detailed terms.
- Design abstraction – description of structures, algorithms, components, interfaces, etc.
- Implementation abstraction – implementation description which is done in some specific programming language.

As it has been already said, the starting point of re-engineering is existing source code of some application. The process of re-engineering finishes with target software source code as its final result. This process can be more or less complex. For example it can only translate the software from one programming language into another, it can enhance some characteristics or redesign the whole application, etc.

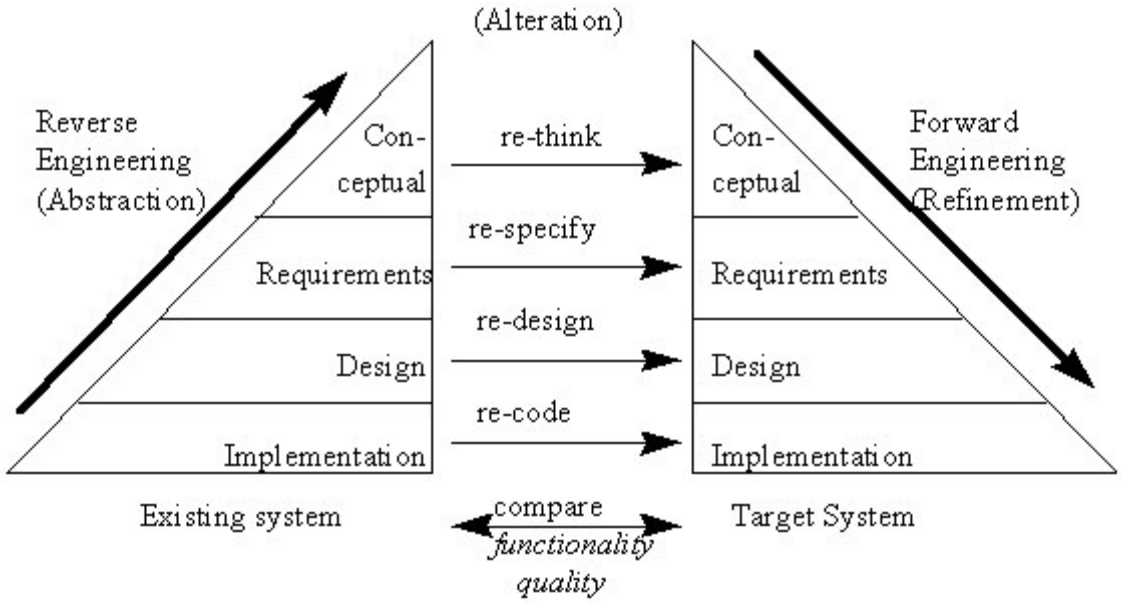

Fig. 1: General Model for Software Re-engineering

The model in Figure 2 applies three principles of re-engineering: abstraction, alteration and refinement (types of change in stages of software development (Byrne, 1992)).

Abstraction is a gradual increase in abstract level of software. Software representation is created by successive replacement of existing detailed information with information that are more abstract. This is termed reverse engineering. Refinement is gradual decrease in the abstraction level of software representation and is caused by successive replacement of existing software information with more detailed information. This is termed forward engineering and resembles software development of new code, but with some process refinements. Alteration may have two dimensions; either as a change of functionality, or as a change of implementation technique (e.g. development technology) (Jacobson & Lindstrom, 1991) .

If we want to change some software characteristics we have to do changes at abstraction level where information about those characteristics are explicitly stated. If we want to simply translate code into another programming language, we do this (alteration) at the implementation level. No reverse engineering is needed here. With the increase of abstraction level, the alteration tasks change and also the need for some tasks of reverse engineering change. If we want to re-specify some requirements reverse engineering must me applied to the implementation and design in order to get the functional characteristics.

Reverse engineering is the process of analyzing software to identify its components and relationships between them and to create representations of the system in some other form or at a higher level of abstraction (Rosenberg, 1996).

3. Re-engineering approaches

There are 3 different approaches in re-engineering and they mostly differ by the amount and rate of replacements that are made in existing software to get the target software (Byrne & Gustafson, 1992).

1. Big Bang approach
2. Incremental/Phase-out approach
3. Evolutionary approach

In case of applications generator re-engineering the Incremental/Phase-out approach has been used.

3.1 Incremental/Phase-out approach

Existing system is divided into logical sections and these sections are re-engineered and added to the system as new versions that are needed to achieve certain functionality (Sneed, 2005). In other words, software is dividend into components and those components are being re-engineered. Incremental software re-engineering allows for safer re-engineering, increased flexibility and more immediate return on investment (Olsem, 1998). One example of incremental re-engineering is incremental transformation of procedural systems to object oriented platforms in which a generic re-engineering source code transformation framework is used (Ying & Kontagiannis, 2003).

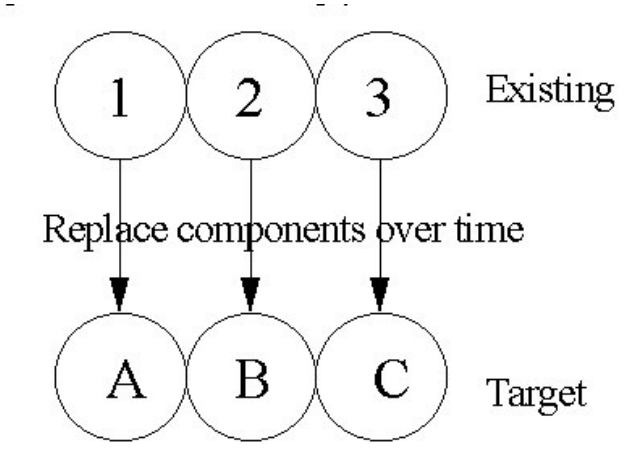

Fig. 2: Incremental/Phase-out Re-engineering Approach

3.2 Hybrid Re-engineering - COTS Track Hybrid Re-engineering

Another approach to re-engineering is hybrid approach (Ajlouni & Hani, 2006). There are many variations in this approach and COTS Track Hybrid Re-engineering is one of them. This approach also relates to applications generator re-engineering. In the COTS track of Hybrid re-engineering, shown in Figure 3, requirements and functions that can feasibly be implemented using COTS must be identified (Rosenberg, 1996).

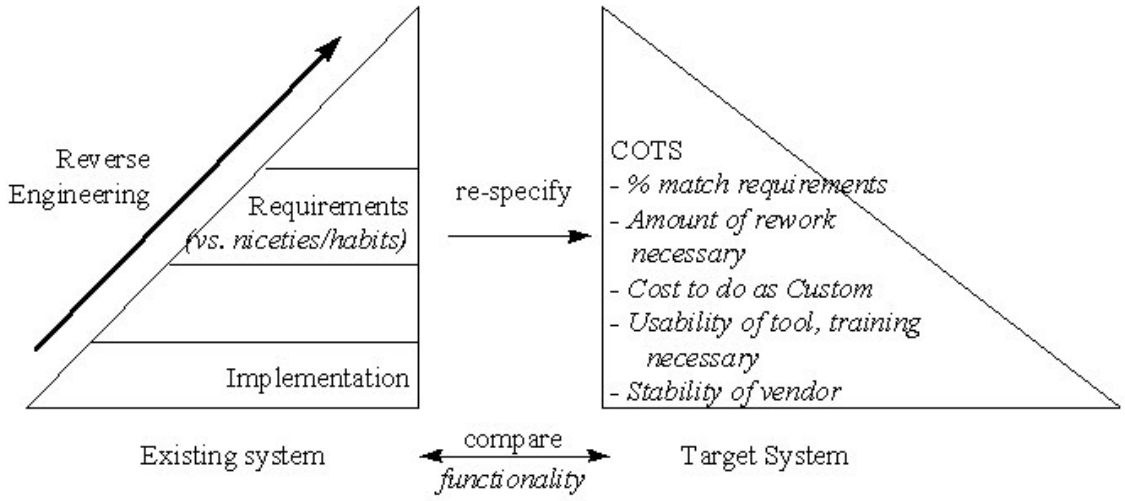

Fig. 3: COTS Track Hybrid Re-engineering

After the reverse re-engineering has been done to identify the requirements, it is very important to separate those requirements that must be contained in the target system ("necessary") from those requirements that users want in the new system because they have become habits or if users are used to them ("nice"). This separation is critical to COTS selection. The advantage of using COTS is in decreased development time and increased reliability (Ajlouni & Hani, 2006). Evaluation, testing and comparing target system with current system must also be performed.

4. Re-engineering of generators

4.1. Re-engineering phases and tasks

Re-engineering consists of 5 phases (Rosenberg, 1996) but for the process of application generator re-engineering only 3 phases are emphasized.

These phases are:

1. Re-engineering team formation
2. Project feasibility analysis
3. Analysis and planning
4. Re-engineering implementation
5. Transition and testing

4.1.1. Analysis and planning

This re-engineering phase has three steps: analyze the current system, specify the characteristics of the target system, and create a standard test or validation suite to validate the correct transfer of functionality (Rosenberg, 1996).

In first step the existing system has to be described and understood. We use manuals, documentation, code and any other usable source that could help us in understanding of the current system.

In second step we have to define metrics that will help us in assessment of current system and its characteristics and also to define characteristics that have to be improved, priorities, quality problems, all according to technical and business values. Metrics and assessment have to be used until the very end of re-engineering process in order to monitor all consequences of every single change, that is, to monitor its impact on the system.

Finally, a standard test and validation suite must be created. These will be used to show that the new system is functionally equivalent to the current system and to demonstrate that the functionality is unchanged after re-engineering.

4.1.2. Re-engineering implementation

In this phase reverse engineering is used to describe the current system at a desired level of abstraction. After this, forward engineering is used. Forward engineering can be compared to standard software development process. The goal is to redesign the system to fit new goals. Validation and measurement of progress and effects must also be performed in order to assess the improvements and to find potential problems and risks.

4.1.3. Testing and transition

Testing is important to determine effects and functionality errors in the target system after re-engineering. The same tests can be applied to current and target system and they can be compared to see the effect of re-engineering. The documentation must be updated according to changes in the system.

4.2. Analysis and planning of generators

Analysis and planning of generators consists of several phases:

1. separation of concerns
2. forming libraries of characteristics (aspects)
3. forming scripting model of generator

4.2.1. Separation of concerns

Crosscutting characteristics (aspects) are program parts that are not connected to individual organizational program units such as functions and classes, but showing up in various application parts, (Kiczales et al., 1997)(Lee, 2002). Aspects of various application cases are singled out into application specification, i.e., separation of concerns (views) is done as presented by (Stein et al., 2003). In the following example, some parts of the code are specific for the particular program (shown in grey), and some are common for all programs from the same problem domain:

```
#include <iostream.h>
int first;
float second;                    data declarations
char third[40];
void main(){
//entry of values
cout << "first = ";
cin >> first;
cout << "second = ";             data entry
cin >> second;
cout << "third = ";
cin >> third;
// processing -forming the list of fields
cout << "Lista polja:first,second,third"; list of fields
// console output of values
cout << endl << "--------------------" << endl;
cout << "first = ";
cout << first << endl;
cout << "second = ";             data output
cout << second << endl;
cout << "third = ";
cout << third << endl;
}
```

Common parts of program forms main metaprogram:

```
#include <iostream.h>
#fields#
void main(){
//entry of values
#entry#
//processing –forming the list of fields
#processing#
//console output of values
cout << "-------------------" << endl;
#output#
}
```

The data *declaration part* is replaced by tag *#fields#*, *data entry* by *#entry#*, *list of fields* by *#processing#* and *data output* by *#output#*. Now, the same process should be done on each part, replaced by replacement tags, for example:

```
int first;
float second;                    data declarations
char third[40];
```

This is kind of repetition, because there are three declarations like following:

<*type*> <*variable*>

It could be solved by different metaprograms for each type of variable:

field_number:

```
int #field_number#;
```

field_real:

```
float #field_real#;
```

field_char:

```
char #field_char#[40];
```

4.2.2. Application specification

It's easier to form the application specification than metaprograms, because application specification consists only from specific properties (aspects) of particular application. These aspects occur in different parts of application. For example, such properties in observed program are variables *first*, *second*, and *third*. Program deals with that variables (and surrounding text) in all of four main parts (*declaration part, data entry, list of fields* and *data output*). Extracted specific properties are in hierarchic order, where higher levels define groups and repetitions. Such relationships are shown in the specification diagram (Radošević, 2005)(Fig. 4).

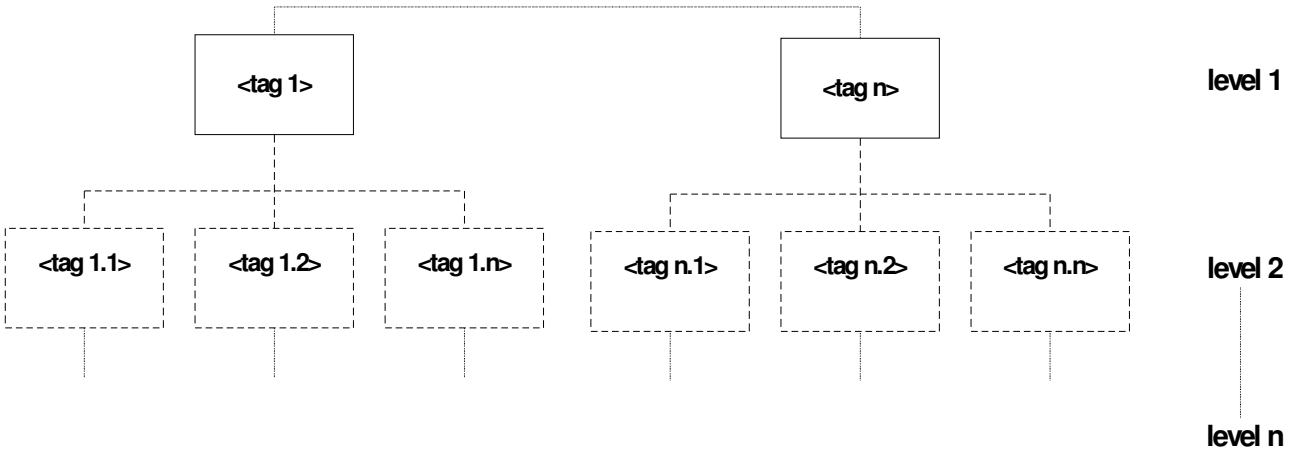

Fig. 4: The specification diagram

In observed example, the specification diagram is quite simple (Fig. 5):

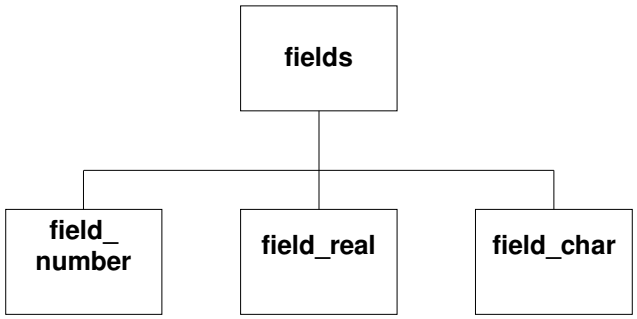

Fig 5: The specification diagram of the observed example program

The application specification is in textual form, and is defined by the specification diagram. Such specification of the observed program is as follows:

fields:
field_number:first
field_real:second
field_char:third

4.2.3. The metascripts diagram

The metascripts diagram (Radošević, 2005) defines connection of specification elements to metaprograms. The structure of diagram is defined by hierarchy of metaprograms. Metaprograms are mutually connected by links (replacement tags in metaprograms). Each link contains data source (sources are defined in the specification diagram). Elements of the metascripts diagram are shown in Fig. 6:

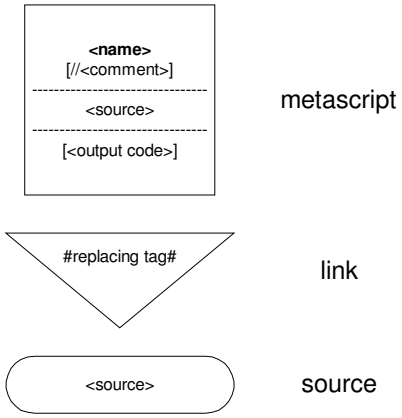

Fig. 6. Elements of the metascripts diagram

The metascripts diagram structure is shown in Fig. 7:

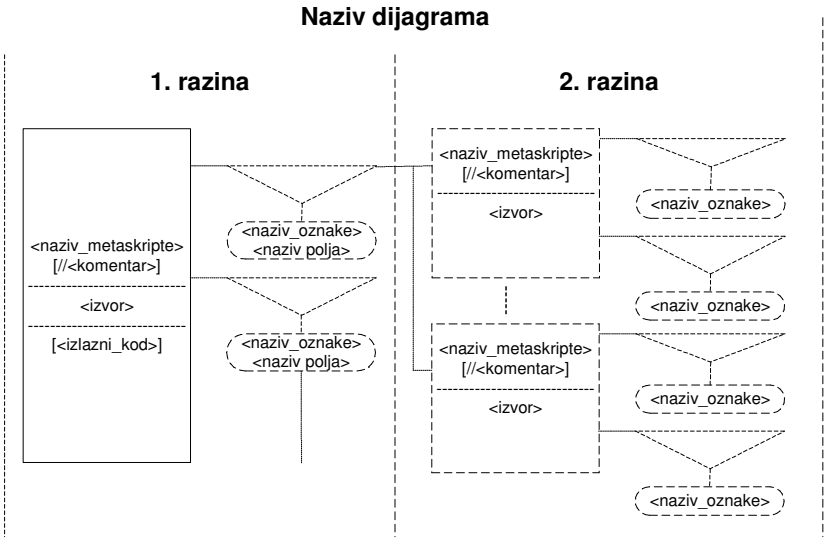

Fig. 7: Stucture of metascripts diagram

The particular diagram for the observed program is shown in Fig. 8:

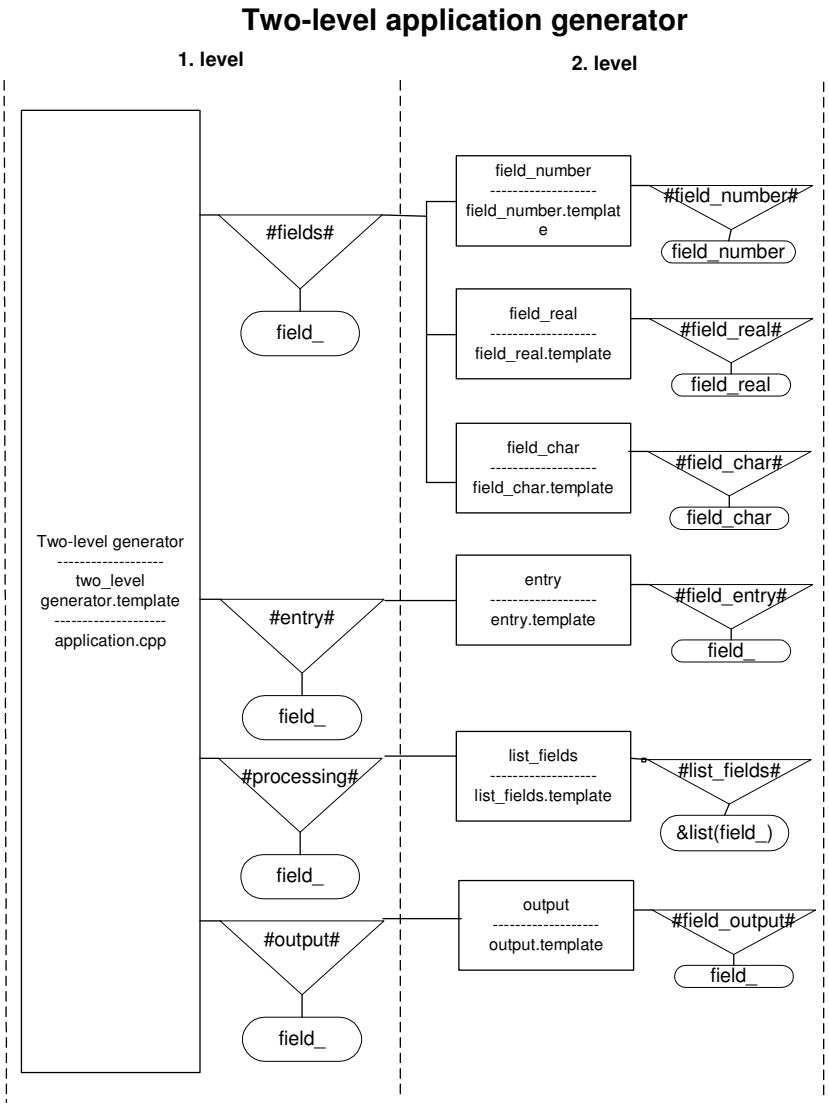

Fig. 8: The metascripts diagram of two-level application generator from the example

First level of the metascripts diagram shows the application in a whole, with main metaprogram and it's replacement tags (here called as links). Second level deals with main parts of observed program. Sources in first level are defined as groups (names ends with "_" sign), while sources on the second level are connected to specific specification elements on the second level of specification diagram (Fig. 5).

4.3. Re-engineering implementation

For the purpose of generators development, the C++ library was made (Radošević, Orehovački & Konecki, 2007.). That library enables implementation of generators based on scripting model, through generative objects. Generative objects are objects from classes which are included in programs in a form of libraries. For that purpose, the appropriate library for C++ was developed.

4.3.1. C++ library for generator development

The library defines two classes for generator development: *cgenerator*, and *cspecification*. The *cgenerator* class enables implementation of generating functions, while the *cspecification* class inherits *cgenerator*, adding methods for working with application specification.

4.3.2. Class cgenerator

The *cgenerator* class enables implementation of simple one-level generator in C++ language, which is shown in the next diagram (Fig. 9).

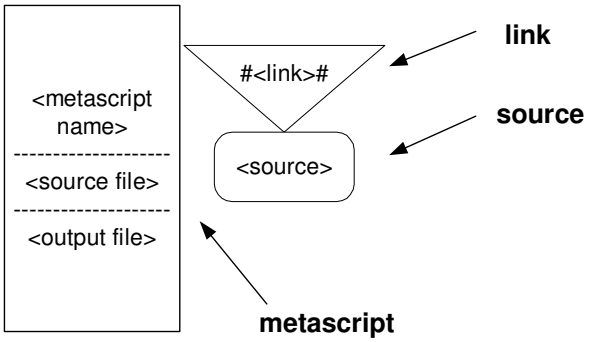

Fig. 9: Single level generator

Operations supported by appropriate methods from *cgenerator* class are following:

- loading program code templates (metascripts)
- simple generating by exchanging links using appropriate exchange contents (sources)
- saving generated program code into output file
- different operations on character strings, like concatenation of generated code and assembling templates

4.3.3. Class cspecification

The *cspecification* class enables working with application specification. Application specification is proposed by specification diagram (Fig. 3). The *cspecification* class inherits *cgenerator* and enables implementation of specification linked list, all operations connected to application specification and implementation of more complex generating functions. The application specification is in a simple textual file, in a form of label-value pairs, like the following example:

title:students
field_int:id
field_char:surname_name
field_float:average_mark

The linked list of application specification is formed by loading from textual specification (Fig. 10).

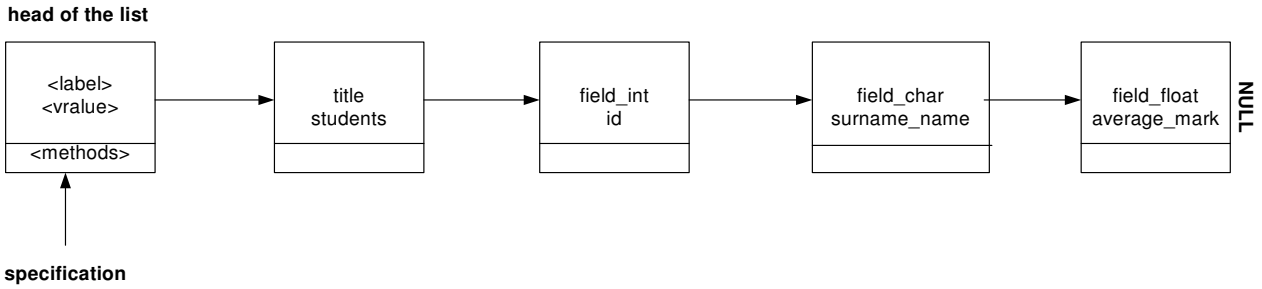

Fig. 10: Linked list of specification

Operations supported by appropriate methods from *cspecification* class are following:

- loading specification to specification linked list
- implementation of simple single level generator (Fig. 9)
- selecting parts of specification, due to proper connecting sources to metascripts.

4.3.4. The structure of generator

The general structure of generator based on C++ generative objects is shown in Fig. 11:

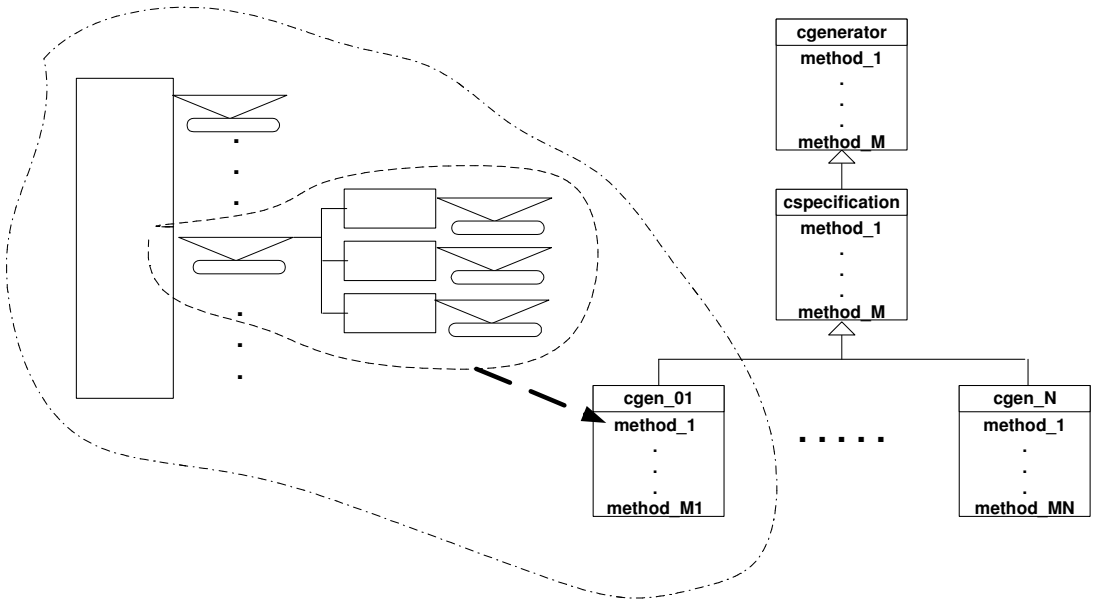

Fig. 11: General structure of generator based on C++ generative objects

As shown in Fig. 11, particular generators are implemented by appropriate classes, which are inherited from *cspecification*. Particular branches of metascripts diagram are implemented by appropriate methods.

4.3.5. Implementation of generator in C++

Generator in C++ uses library that defines classes *cgenerator* and *cspecification* (Fig. 11). The following statements defines working with application specification and the main metascript:

```
// cspecif1 inherits cspecification
  cspecif1 *specification=new cspecif1;
  // loading specification
  specification->load("generator_cpp.specification");
  // loading metascript
  specification->metascript("two_level_generator.template");
  // generirating code
  specification->generating_script();
  // saving generated code
  specification->save(output_filename);
```

Class *cspecif1* is used for generating program code:

```
class cspecif1:public cspecification{
 public:
 void generating_script (){
  cspecification *current=this->next;
  //title and name od the table
  char colector[3000]="";
  current=this-> next;
  while (current){
   easy_generator(current,"","field_int","# field_int #",NULL);
   easy_generator(current,"","field_float","# field_float#",NULL);
   easy_generator(current,"","field_char","# field_char#",NULL);
   current=current-> next;
 }//while
. . . . .
. . . . .
};//cspecif1
```

The method *easy_generator* is used for implementing a single one-level generator (Fig. 9). Generating starts with reading the specification and specifying the exchange of replacement tags (marked with **#** sign) by values from specification which is involved in *easy_generator* method.

4.4. Generator and application maintenance

The whole process of generator and application maintenance could be shown in next diagram, according to Boehm spiral model of software development (Boehm, 1988) (Fig. 12).

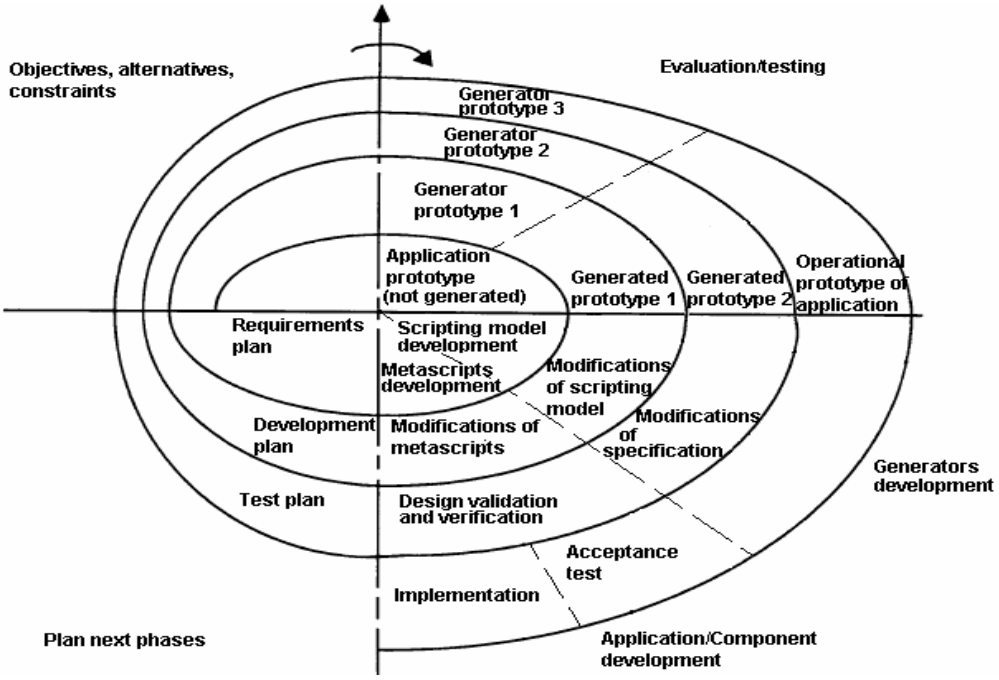

Fig. 12: Generative application development as spiral development using the Boehm (Boehm, 1988) model

Generative application development begins with the Requirements plan and the problem domain prototype application. This is followed by the Separation of concerns, so that specific characteristics of each individual application are contained within its specification, while common characteristics end up in metascripts. The scripting model defines the assembling of given application within the presented problem domain.

5. Conclusion

It is shown in this paper that scripting model of generators can be implemented by appropriate object model by using some software reengineering approaches, like incremental model and hybrid COTS. For that purpose, the appropriate library for C++ was developed, as well as an example of C++ source code generator. Development of applications and their generators is adapted to Boehm's spiral model of software development (Boehm, 1988). It also shows the whole process of generator development, started from prototype program reengineering through separation of concerns, making scripting model of generator and generator implementation through generative objects.

6. References

Ajlouni, N. & Hani, F. B. (2006). Redesigning legacy systems using hybrid re-engineering, *International Conference on Information & Communication Technologies: from Theory to Applications*, pp. 2784- 2785, ISBN: 0-7803-9521-2, Damascus (Syria), 24-28 April 2006, IEEE Computer Society Press

Boehm, B.W. (1988). A Spiral Model of Software Development and Enhancement, *Computer*, Vol. 21, No. 5, May 1988, pp. 61-72, ISSN: 0018-9162

Byrne, E. J. & Gustafson, D. A. (1992). A software re-engineering process model, *Proceedings of the Sixteenth Annual International Computer Software and Applications Conference*, pp. 25-30, ISBN: 0-8186-3000-0, Chicago (USA), 21-25 September 1992, IEEE Computer Society Press

Byrne, E. J. (1992). A conceptual foundation for software re-engineering, *Proceedings of the Conference on Software Maintenance*, pp. 226-235, ISBN: 0-8186-2980-0, Orlando (USA), 9-12 November 1992, IEEE Computer Society Press

Jacobson, I. & Lindstrom, F. (1991). Re-engineering of old systems to an object-oriented architecture, *Conference proceedings on Object-oriented programming systems, languages, and applications,* pp. 340-350, Phoenix (USA), 6-11 October 1991, Vol. 26, No. 11, ISSN: 0362-1340

Olsem, M. R. (1998). An incremental approach to software systems re-engineering, *Journal of Software Maintenance: Research and Practice*, Vol.10, No.3, May 1998, pp. 181-202, ISSN:1040-550X

Radošević, D. (2005). Integration Of Generative Programming and Scripting Languages. *Doctoral thesis*, Fakultet organizacije i informatike, Varaždin, 2005.

Radošević, D.; Orehovački, T. & Konecki M. (2007). PHP Scripts Generator for Remote Database Maintenance based on C++ Generative Objects. *Submitted to Mipro 2007 conference*, Opatija, Croatia, May 21.-25, 2007.

Rosenberg, L. H. (1996). Software Re-engineering, *Available from:* http://satc.gsfc.nasa.gov/support/reengrpt.PDF, *Accessed:* 2007-03-12

Sneed, H. M. (2005). An incremental approach to system replacement and integration, *Ninth European Conference on Software Maintenance and Reengineering*, Manchester, (UK), pp. 196-205, ISBN: 0-7695-2304-8, 21-23 March 2005, IEEE Computer Society Press

Stein, D.; Hanenberg, S. & Unland R. (2003). Position Paper on Aspect-Oriented Modelling: Issues on Representing Crosscutting Features, *International Conference on Aspect-Oriented Software Development*, Boston (USA), 17-21 March 2003

Ying, Z. & Kontagiannis, K. (2003). Incremental transformation of procedural systems to object oriented platforms, *Proceedings of 27th Annual International Computer Software and Applications Conference*, ISBN:0-7695-2020-0, Dallas (USA), pp. 290-295, 3-6 November 2003, IEEE Computer Society Press.

Corresponding Author Data:

Name and email address of corresponding author: Danijel Radošević, danijel.radosevic@foi.hr

Manuscript Data:

1. **Author(s) Name(s): Danijel Radošević, Tihomir Orehovački, Mario Konecki**
2. **Title of Manuscript:** Generator development through reengineering process
3. **Key words: generative programming, generative objects, software reengineering**
4. **Abstract:** Development of scripting model based generators is re-engineering process, which consist of several phases: making application prototype, defining specification elements and code templates (metaprograms) through separation of concerns, making generator scripting model and generator implementation through generative objects using appropriate C++ library. That process corresponds to Barry Boehm spiral model of software development. Main benefits of that approach are flexibility in generator development and easier maintenance of generators and generated applications.
9. **Please send my copy/copies of Book to the following address:** Faculty of organization and informatics, Pavlinska 2, 42000 Varaždin, Croatia

All Sc Book Authors Data:

1. First / Middle / Family Name: Danijel Radošević
2. Academic Titles: PhD
3. Position / Since: Assistant Professor/2006
4. Institution: Faculty of organization and informatics, University of Zagreb
5. Place, Date and Country of Birth: Zagreb, Croatia, 1969-03-27
6. Nationality / Citizenship: Croatian / Varaždin
7. Field of interests: generative programming, text mining
8. E-mail address: danijel.radosevic@foi.hr
9. Site: http://www.student.foi.hr/~darados
10. Phone & Fax #: 385 042 390 834, 385 042 213413
11. Postal address: Pavlinska 2, Varaždin, Croatia

1. First / Middle / Family Name: Tihomir Orehovački
2. Academic Titles: BSc
3. Position / Since: Teaching Assistant/2006
4. Institution: Faculty of organization and informatics, University of Zagreb
5. Place, Date and Country of Birth: Koprivnica, Croatia, 1982-08-24
6. Nationality / Citizenship: Croatian / Čakovec
7. Field of interests (key words): generative programming, AI, semantic web
8. E-mail address: tihomir.orehovacki@foi.hr
9. Site: http://www.foi.hr/nastavnici/orehovacki.tihomir/index.html
10. Phone & Fax #: 385 042 390 853, 385 042 213 413
11. Postal address: Pavlinska 2, Varaždin, Croatia

1. First / Middle / Family Name: Mario Konecki
2. Academic Titles: BSc
3. Position / Since: Teaching Assistant/2006
4. Institution: Faculty of organization and informatics, University of Zagreb
5. Place, Date and Country of Birth: Daruvar, Croatia, 1982-08-03
6. Nationality / Citizenship: Croatian / Varaždin
7. Field of interests (key words): generative programming, web technologies, e-learning
8. E-mail address: mario.konecki@foi.hr
9. Site: http://www.foi.hr/nastavnici/konecki.mario/index.html
10. Phone & Fax #: 385 042 390 860, 385 042 213 413
11. Postal address: Pavlinska 2, Varaždin, Croatia