Cumulant – a program facilitating the development of educational projects series stepwise cumulating the functionality of the developed program

Rudolf Pecinovský

ICZ a.s., Na Hřebenech II 1817, 140 00 Praha 4
University of Economics Praha, W. Churchill sq. 4, 130 67 Praha 3
rudolf@pecinovsky.cz

Abstract. The paper introduces a program facilitating the development of series of educational projects, in which the developed program is stepwise improved and at the same time, each phase is presented as an independent project assigned to the relevant lesson of the textbook or course. It explains how to specify assembling of particular projects from classes in the overall teacher's project and shows which tools the program offers for defining several versions of the gradually developed class in one source file, and how to specify which part of the source code will be incorporated into the final program for particular lessons.

1 Introduction

When we prepare textbooks or courses of programming, we accompany them by a set of programs that serve for demonstration of the explained topics or for exercises in which students show how they understand the explained topic. As indicated in [2] the accompanying programs can be designed according to several concepts. The best (but also the most laborious) is the concept, according to which you demonstrate the thesis on few (preferably on one) of the stepwise developed and improved projects that are modified by teacher as well as by students.

This concept is laborious due to the fact that we have to keep the mutual consistence of particular versions of the developed programs. It can happen during the development that when developing the later lessons we have to change also the source codes of previous versions. Such correction logically affects all versions starting by the oldest one, just developed.

The consequence of these unpleasant characteristics is that in case the teacher would decide to supplement his/her textbooks or courses by such accompanying programs, he tries to change them only by adding whole source files. If he/she needs to modify one of the previously defined files, he/she tries to only add a new method.

The *Cumulatnt* application introduced in this paper can help with solving majority of unpleasant accompanying effects of programs designed within this concept.

2 Overall concept

The *Cumulant* application is constructed as a self-extracting archive containing source codes of the teacher's project and if need be further files. Then, on demand it generates from them the source codes of the required versions of accompanying projects.

As the teacher's project incorporate all files of all generated projects, the documentation names it a *monolith*. This is how it will be named further in this paper.

The generating of required versions of accompanying programs runs in two phases:

- In the first phase the generator checks up from which monolith's files the separate required projects should be assembled and how these projects should be organized in packages.
- In the second phase the generator converts particular files into developed projects, whilst each of the source files is processed by a preprocessor that modifies the final stage to a shape corresponding with the destination project.

The whole generator including the monolith's source files is located in JAR archives where typically two virtual folders are created:

- Folder `AUX` containing the file `Projects.txt` with information for controlling the required project generation and possibly other auxiliary files. One of them is the `Basic.properities` file, where certain key information concerning the planned generation is placed. Among them the property `rrf` plays an important role specifying the relative root folder – the common parent folder of all files in the monolith. Addresses of all rendered files can be specified in relation to this folder.
- The `SRC` folder containing all source files of the monolith. This folder is optional, because all data can be placed in the `AUX` folder, but from practical reasons it is convenient to place the monolith's source files here, because we can simply copy them here from their original location.

3 The control file `Projects.txt`

A very simple DSL (Domain Specific Language) has been created for specification of the content of particular target projects. Theoretically we could possibly use the XML or JSON formats, however both are in vain "talkative", because they require specification of each entered value despite it can be derived from its position in the text.

Syntax of this control language is simple, its description in EBNF is in the listing 1. The language is line-oriented – each statement is on a separate line. The program reads this file line by line, interprets the statements and passes the preprocessed information to objects that are involved in preparing basis for the subsequent construction of projects. The rules for individual lines of the control language are as follows:

- The empty lines are ignored.
- The lines beginning with the currency symbol "¤" are taken as commentary and are ignored; in different locations the currency symbol is used as separator.
- In the lines starting with `PROJECT` ¤ the text after the separator is interpreted as a name of the project and at the same time of the folder containing this project.

- In the lines starting with `PACKAGE` ¤ the text after the separator is interpreted as a package name which means it specifies the name of the target folder, where the files from this package should be saved.
- In lines starting with `FOLDER` ¤ the text after the separator is interpreted as a path to a source folder incorporating all other files until it will be set otherwise (i.e. the source file name will be assembled in a different way).
- In the lines starting with `BLOCK` ¤ the text after the separator is interpreted as a block name which is a group of classes that should be included into more projects. Block has a similar structure as a project and it serves as a container for a set of files and folders that should be included in several projects and so it is useful to define it ahead.
- In the lines starting with `INCLUDE` ¤ the text after the separator is interpreted as the name of the block which should be included in a given place into the currently created project.
- The line starting with `END` marks the last input line. All lines after it are ignored. This means the author can save here information useful for a future use.
- If none of the previous rules is valid, the text placed left towards the first separator is considered as the destination file name and the text in the right as the name of the source file. If there is no name of the source file, we suppose it is the same as the destination file name.

  If the text contains at least two separators, the text behind the second one is interpreted as the project ID, into which the source file will be rendered during the next step.

The above description is not the detailed user documentation. It only tries to show that the application supports even many nonstandard needs that may appear during preparation of such kind of accompanying/additional programs.

Listing 1: Language syntax specifying the content of separate projects

```
Program = { Block | Project }
BPBody  = { Package }
Block   = "BLOCK"  "¤"  BlockName  [ "¤" Comment ]
          "\n"  BPBody
Project = "PROJECT"  "¤"  ProjectName  [ "¤" Comment ]
          "\n"  BPBody
Package = "PACKAGE"  "¤"  [ PackageName ]  "\n"
          { Folder  |  Include }
Folder  = "FOLDER"  "¤"  [ ">P"  |  SourcePath ]
          "\n"  { File }
Include = "INCLUDE"  "¤"  BlockName  "\n"
File    = { DestFileName  "¤"
                          ( "¤"  |  SourceFileName )
                          [ "¤"  ProjectID ]  "\n"
```

4. Organization of source files

The previous chapter described how we can specify which files should be included in particular versions of developed projects. Beside it we need also to specify how the particular source files will be rendered from the monolith to the destination projects files.

The basic problem of such rendering is, that majority of the source files appear in several – gradually improved forms during the project development. It is not suitable to have a different version of the source file for each shape because otherwise, big problems connected with keeping their mutual consistence arise. On the other hand it is also not suitable to specify all modifications into one file, because the file becomes then totally confused and prone to be contaminated by various errors.

However, if we split the definition sequence into several files, we have to solve the problem with their names. The Java public classes should have the same name as their source file. We could assign them into separate packages, but the number of packages then might unpleasantly increase.

The *Cumulant* application utilize that the above mentioned rule about the public class names is in force only for top level classes. When we define the class as the nested one ([1]), we need not keep any special rule for its name. So the recommended solution is to nest the modified class each time into class with name consisting from the name of its nested class followed by an information about the range of lessons, the relevant source code is determined for them. In addition the document comments of the wrapping top classes can contain information about differences of particular versions of the destination classes rendered from the inner one and facilitate so the possibly future modifications.

5. Rendering of the source files into the destination form

The application uses another DSL – it allows defining how to render one correct java source file into several versions of the destination files. *Cumulant* uses for this purpose special preprocessor line comments. They are as follows:

`//%%+`

It concludes the preliminary section containing lines that will not be copied into destination file.

`//%A+`

It serves as opening quotation of line sequence that will be added into the destination file only when arguments of this statement are evaluated as `true`. The group have to be closed with the preprocessor comment `//%A-`.

`//%I+`

It serves as opening quotation of sequence of commented lines. These lines will be uncommented and added into the destination file when arguments of this statement

are evaluated as `true`. Otherwise the group will be omitted. This group should be closed with the preprocessor comment starting `//%I-`.

`//%X+`
It serves as opening quotation of line sequence that will be excluded from copying into the destination file when arguments of this statement are evaluated as `true`. Otherwise the lines from this group will be copied into the destination file. This group should be closed with the preprocessor comment starting `//%A-`.

`//%%-`
It opens the final section containing lines that will not be copied into any destination file.

6 Parameters of preprocessor comments `//%X+`, `//%I+` a `//%X+`

Arguments of preprocessor comments specify the set of projects, for which the statement should be activated. They should be written in the following form (the symbol `4P` states for first 4 characters of the project name representing the project ID):

`<4P` projects with ID less than `4P` will be added into selected set

`<=4P` projects with ID less than `4P` or equal will be added into selected set

`=4P` projects with the given ID will be added into selected set

`>4P` projects with ID greater than `4P` will be added into selected set

`>=4P` projects with ID greater than `4P` or equal will be added into selected set

`-4P` projects with ID greater than `4P` or equal will be removed from the selected set (projects with ID up to `4P` remain in the set)

`-=4P` projects with ID greater than `4P` or equal will be removed from the selected set (projects with ID up to `4P` or equal to `4P` remain in the set)

Arguments must be entered in an order corresponding to increasing ID. The preprocessor comment

```
//%A+ >105 -110  >=115
```

opens a group of lines that will be included into projects with ID from `105` till `110` (excluding these two) and into projects with ID greater or equal to `115`.

7. Summary

This paper introduced a program *Cumulant* that may significantly facilitate the preparation of accompanying programs to programming textbooks and courses in which the designed program is stepwise improved, and each advancing version belonging to a lesson or to a part of a lesson is published as a separate project. Thus the students can follow individual steps and in case of problems appearing during development of such a group of projects they can get back to the previous step. The paper has shown how these problems can by solved using the introduced program.

Acknowledgement

This work has been created with a kind support of the grant *The Research of methodology how to teach programming and its possible improvement* announced by the RPF foundation.

Literatura

[1] GOSLING James, JOY Bill, STEELE Guy, BRACHA Gilad: *The Java Language Specification Third Edition*. Addison-Wesley 2005. ISBN 0-321-24678-0

[2] PECINOVSKÝ Rudolf: *Programming Textbooks and Courses.* Proceeding of Objects 2011. ISBN 978-80-554-0432-5.