# STRATIFIED PROGRAMMING: TOWARDS A NEW PARADIGM FOR SOFTWARE DEVELOPMENT

**Adrian Pasculescu**
**Alpas Solutions**
**Toronto, Ontario**
**Canada, L5C 1Y1**
**adrianp@alpas.net**

**Sergiu Dascalu**
**Department of Computer Science**
**University of Nevada, Reno**
**Reno, NV, 89557, USA**
**dascalus@cs.unr.edu**

**Abstract**

This paper introduces stratified programming, a novel approach for program construction. In essence, stratified programming allows the developer to build and execute software at various levels of abstraction, each level corresponding to a program stratum that provides a specific degree of functionality. Although there is a significant amount of work reported in the scientific literature on program refinement none of this work, to the best of our knowledge, has proposed mechanisms for switching the level of software details at the execution stage or suggested the use of program strata in a larger software engineering context. This not only makes the approach described in this paper highly innovative but also opens new related research and development directions across the entire software life-cycle. To illustrate the approach we present an example focused on the coding phase of the software process. Several avenues of further investigation are also indicated and the practical benefits of stratified programming are discussed in the context of modern software construction methodologies.

**Keywords**: stratified programming, program strata, layered execution, strata-based software construction.

## 1 INTRODUCTION

Abstraction is a fundamental principle applied for solving problems, including problems pertaining to the software engineering domain [1, 2]. Abstraction is essentially an instrument for dealing with complexity: it allows us to focus on the most relevant aspects of the problem and also, by changing the level of abstraction, to focus on the most relevant aspects of a part of this problem. Refinement, involving "changing the level of abstraction," is probably the most general and powerful technique for handling complexity, a technique deeply ingrained in our cognitive profile. High level of abstraction means "general," "fundamental," "most important," "on a larger scale," while low level of abstraction means "detailed," "on a smaller scale," and sometimes (but by all means not always!) "less important". High level of abstraction means seeing the forest on the horizon line while low level of abstraction means looking from few yards at one of that forest's tree or analyzing under the microscope the delicate nature of the tree's leafs.

Abstraction and refinement are embedded in practically all techniques and tools used in software engineering. Examples of such techniques are the top-down approach for software development [3], the definition of classes and patterns in the object-oriented technology [4], and the successive refinements of formal specifications [5]. Tools using various levels of abstraction and providing mechanisms for switching the level of detail include file and document management facilities, in which folders and document contents can be seen either "collapsed" or "expanded" (e.g., XML documents [6]); versioning control systems such as the *Concurrent Version System* (CVS) [7] or Microsoft's *Visual Source Safe* [8] where, based on incremental updates, various stages of a project are preserved; and specialized editors such our own *Harmony* environment for combined use of semi-formal and formal notations in software specification, where the level of on-screen detail of formal specifications can be adjusted [9].

Although a significant amount of work has been focused on program refinement (e.g., [10]) none of this work has either proposed mechanisms for switching the level of software details at both the build and execution stages or suggested the use of program strata in a larger, pragmatic software engineering context. In this paper we introduce *stratified programming* (SP), a software development approach based on the notion of *program strata*. Each *program stratum* corresponds to a *layer of functionality* assigned to software, higher the position of the layer in the strata hierarchy, more general (more abstract) the functionality provided. Each program stratum incorporates the functionality of the strata above it and provides some additional, specific functionality.

The idea of stratified programming has emerged from our practical software development experience and has been driven by the goal of accelerating software production. Thus, we have focused first on the coding stage and have devised a flexible yet systematic way for building programs incrementally. This paper presents the use of program strata at the coding stage, yet a more precise description would be "between unit design and unit testing," with coding activities placed in the centre of this "window" of the software process. However, SP is not only about unit design, implementation, and testing.

While we agree with Beck that "at the end of the day, it has to be a program" and "nominate coding as the one activity we know we cannot do without" [11] we are aware that building software well transcends the coding phase of the software process [12]. In short, driven by pragmatic considerations that demand rapid software production we look in this paper at stratified programming at the coding (implementation) phase, but see implementation only as the starting point from which several new research directions related to *strata-based software construction* (SSC) emerge. While focusing on coding places our approach under the umbrella of new software methods aimed at increasing the productivity of software construction (e.g., agile development [13]), applying the strata-based approach to other phases of the software process, including requirements specification and software maintenance, will give our approach the broader scope of the entire software life-cycle.

We believe that our approach offers an innovative, pragmatic solution for software development and opens a number of interesting avenues for further investigation. The closest approach to ours is probably program refinement, which starts with a software specification at the highest level of abstraction and gradually refines it into more and more detailed descriptions of the software, possibly up to an executable program. Our approach, also incremental in nature, is nevertheless distinct in that it makes use of program strata during the entire development cycle, including run-time execution. This, we believe, provides significant flexibility to both software development and software execution.

This paper, in its remaining part, is structured as follows: Section 2 presents a very detailed, albeit simplified example of stratified programming, Section 3 reviews the concepts behind SP and evaluates its merits and limitations, Section 4 identifies directions of further research and development, and Section 5 concludes the paper with a brief summary of our contributions.

## 2 AN EXAMPLE

### 2.1 The Problem

To illustrate the concepts of program strata and stratified programming, let us consider the following simplified example (Fig. 1) in which a log file from a database management system has to be analyzed in order to detect occurrences of specific events. The log file has an optional header and a number of fixed-format sections, each section starting from the beginning of the line with a number followed by ")". In Fig.1 these sections begin, in order, with "3)", "19)", "20)", etc. Within each section, paragraphs are identified by their names, for example Client Network Name, Type, Start Time, Stop Time, etc. The goal of analyzing the log file is to identify sections that include a Text paragraph and extract from them their Text, Start Time, and Stop Time paragraphs.

```
EVENT LOG HEADER
   Event Monitor name: PERFDB_MON
   Database Name: PERFDB
   First connection timestamp: 09-05-2002
   10:37:48.255422
   Event Monitor Start time:   05-09-2002
   15:13:48.700463
  ------------------------------------------
3) Connection Header Event ...
   Client Network Name: unixi1970
   Connect timestamp: 05-09-2002 12:27:40.073004

19)Statement Event ...
   Record is the result of a flush: FALSE
  ------------------------------------------
   Type     :  Dynamic
   Cursor   :  SQLCUR4
   Text     :  SELECT distinct bbp_wire_info_key
               FROM b_w_info WHERE
               (fx_status = ? OR (fx_status = ?
               AND fx_rate_ts <= ?)) AND
               wire_status NOT IN (?,?)
  ------------------------------------------
   Start Time: 05-09-2002 15:13:56.565403
   Stop Time:  05-09-2002 15:13:56.575842
   Rows read:  3887

20)Statement Event ...
   Record is the result of a flush: FALSE
  ------------------------------------------
   Operation:  Static Commit
   Cursor   :
   Start Time: 05-09-2002 15:13:56.578378
   Stop Time:  05-09-2002 15:13:56.578429
   Rows read:  0

22)Statement Event ...
   Record is the result of a flush: FALSE
  ------------------------------------------
   Type     : Dynamic
   Operation: Prepare
   Text     : SELECT DISTINCT contract.id FROM
              user, contract, seclink WHERE
              user.id = ? AND user.contract_key
              = contract.contract_key AND
              (user.status = '0' or user.status
              = '2' ) AND user.deleted_fl <> 'D'
  ------------------------------------------
   Start Time: 05-09-2002 15:14:02.394358
   Stop Time:  05-09-2002 15:14:02.395116
   Rows read: 3
```

**Fig. 1 Log File from a Database Management System**

For illustration purposes, we assume that the input log file comes from the standard input and that the coding of the *Log File Analyzer Program* (LFAP) is done in Perl. The following shows the step-by-step creation of LFAP's strata.

### 2.2 Creating Strata

**Step 1**: The first stratum is defined (Fig. 2). The role of this program stratum is to do nothing; no input is necessary and no output is produced. This might seem odd, but invariants play an important role in mathematics-

based techniques, including formal software specification. In this example, the general context is declared (Perl used for coding, strict syntax checking selected) and the notation rules imposed by the developing company are followed (input and output specified, development phase and date indicated). The net result of this unit's testing will be a valid program that does nothing.

```
#!/usr/bin/perl -w
use strict;
#######################################
#  input : STDIN (DB2 log file)
#  output: STDOUT (xml document)
#  phase : development
#  date  : 2002-10-25
#######################################
```

**Fig. 2 LFAP: The First Stratum**

**Step 2:** The second stratum is added (Fig. 3). This stratum's simple goal is to read the input file. From the traditional coding perspective the added code should not be indented deeper. However, indentation here is necessary because in SP, by definition*, code written on the same indentation level belongs to the same stratum.* From a testing perspective, it makes sense to differentiate this new stratum because an input file is needed. The expected result should again be nothing but this time without an input file the program will wait forever.

```
#!/usr/bin/perl -w
use strict;

#######################################
#  input : STDIN (DB2 log file)
#  output: STDOUT (xml document)
#  phase : start development
#  date  : 2002-10-25
#######################################

   ## 2nd stratum reads lines from the input
   ## file; no processing
   while (my $line=<STDIN>){
   }
```

**Fig. 3 LFAP: The First and Second Strata**

**Step 3:** The third stratum is created (Fig. 4). In its 3-strata version, the program identifies the first section of the log (in the first while loop) and then reads line-by-line the remaining of file (in the second while loop). As indicated in section 2.1, LFAP's goal is to select from the input file only specific sections of the log, skipping over the optional header of the file. The third stratum does not complete the program, yet it extends the existing LFAP's functionality, allows further testing, and prepares the creation of the next stratum. In fact, it needs certain improvement, as shown in the next step of strata creation.

```
#!/usr/bin/perl -w
use strict;

#######################################
#  input : STDIN (DB2 log file)
#  output: STDOUT (xml document)
#  phase : development
#  date  : 2002-10-25
#######################################

   ## 2nd stratum reads lines from the input
   ## file; no processing
   while (my $line=<STDIN>){
      ### 3rd stratum reads only 'sections'
      ### i.e., it skips over front lines
      ### that do not start with numbers
      ### below, it detects the start of the
      ### first section
      if  ($line =~ /^[0-9]+\)/){ last; }
   }
      ### read line-by-line the rest of the
      ### input log file for processing on
      ### a deeper stratum
      while (my $line=<STDIN>){
      }
```

**Fig. 4 LFAP: Strata One to Three**

**Step 4:** Changes are made to the second and third strata and the fourth stratum is added (Fig. 5). The code shown in Step 3 is inconsistent in that it treats differently the first and the remaining sections of the log file (the beginning of the first log "section" is identified in the first loop, the rest of the log will be processed in the second loop). We take care of this by making the scalar variable $line visible in both loops. This is achieved by declaring the variable line in the second stratum before the first while loop and by removing the reserved Perl word my (which defines the scope of a variable) from references to the variable $line in the conditions of both while loops. An additional change consists in moving the test condition of the second loop from the loops' beginning to its end. Thus, there is a consistent treatment of all sections. In addition to modifications to strata 2 and 3, the fourth stratum (characters in bold) is created. A further step in detailing the functionality of the program, stratum 4 is composed of a line of code that simply generates an output copy of the file. This will be refined in the next steps.

Regarding step 4, we note that changes made in strata 2 and 3 do not affect the expected results (input/output pairs) for each of the previous test cases. More precisely, if we go back to Steps 1, 2, or 3 after making the changes described above we should get the same result. In other words, we have made *neutral code transformations* relative to our stratification criteria, which in this LFAP example is "a specific input/output pair is associated with each stratum". In practice, however, different criteria for building strata can be considered and multiple strata may contribute to the same input/output pair.

```
#!/usr/bin/perl -w
use strict;

#####################################
#  input : STDIN (DB2 log file)
#  output: STDOUT (xml document)
#  phase : start development
#  date  : 2002-10-25
#####################################

   ## 2nd stratum reads lines from the input
   ## file; no processing
   my $line="";
   while ($line=<STDIN>){
      ### 3rd stratum reads only 'sections'
      ### i.e., it skips over front lines
      ### that do not start with numbers
      ### below, it detects the start of the
      ### first section
      if  ($line =~ /^[0-9]+\)/){ last; }
   }
      ### read the rest of the input log file
      ### for processing
      do {
         #### stratum 4: generates output (a
         #### simple copy)
         print $line;
      while ($line=<STDIN>);
```

**Fig. 5 LFAP: The Fourth Stratum and Changes to Strata Two and Three**

**Step 4bis.** Neutral code transformations in stratum four (Fig. 6). With the notion of neutral code transformations introduced, we now proceed to make such transformations to stratum 4 as well. This is shown in Fig. 6, where an if-then-else construct is inserted to bring the program closer to its final goal, the complete identification and extraction of "text" sections. Step 4bis not only further refines the program and shows how a given stratum can be extended ("horizontal" extension of the program) but also illustrates a basic technique used in SP: conditional statements are introduced to generate new strata (they are, in a sense, "markers" for new strata).

**Step 5:** The fifth and final stratum is created (Fig. 7). This final stratum of LFAP contains the code for detecting the targeted "text sections" and for extracting (printing) only their Text, Start Time, and Stop Time paragraphs. Two observations are necessary regarding this stratum. First, the last assignment statement of the stratum ($line="") is somewhat at odds with the classical programming style. Normally, the code for printing the targeted paragraphs would be written using a positive if condition:

```
if ($textDetected) {
   print $line;
}
```

However, this would break our SP ad hoc rule for maintaining the same input/output pair on each stratum. This is why we use the "cancellation" assignment $line="" to reach the end of the second while loop with no alteration of the output. Secondly, the code that prints the desired result is written on a single line, although it includes an if statement:

```
print $fldText.$fldStartTime.$fldStopTime if
($textDetected);
```

Normally, the if statement would open a new stratum. However, to save space, we decided to limit the depth of this program (LFAP) to five strata. However, we believe that LFAP's five strata presented in this paper are nevertheless sufficient to introduce the SP approach and some of its characteristics.

```
#!/usr/bin/perl -w
use strict;

#####################################
#  input : STDIN (DB2 log file)
#  output: STDOUT (xml document)
#  phase : start development
#  date  : 2002-10-25
#####################################

   ## 2nd stratum reads lines from the input
   ## file; no processing
   my $line="";
   while ($line=<STDIN>){
      ### 3rd stratum reads only 'sections'
      ### i.e., it skips over front lines
      ### that do not start with numbers
      ### below, it detects the start of the
      ### first section
      if  ($line =~ /^[0-9]+\)/){ last; }
   }
      ### read the rest of the input log file
      ### for processing
      do {
         #### stratum 4: generates output (a
         #### simple copy)
         if ($line =~ /^[0-9]+\)/ ) {
         #### here we are at the beginning of
          #### a new "section"
         } else {
         #### here we are inside a "section"
         }
         print $line;
      } while ($line=<STDIN>);
```

**Fig. 6 LFAP: Neutral Code Transformations in Stratum Four**

## 3 STRATIFIED PROGRAMMING: DISCUSSION

As shown in the previous example, we base our SP approach one the concept of program strata, each program stratum being associated with an execution layer. It is important to stress this association since otherwise SP may seem to be only a new technique for incremental software development. Because we envisage separate layers of execution, SP is more than that. It means, in essence, that software not only can be developed but also executed at different levels of abstraction. This, we think,

is a powerful paradigm, which has far reaching consequences, as discussed below.

```
#!/usr/bin/perl -w
use strict;
#######################################
#  input : STDIN (DB2 log file)
#  output: STDOUT (xml document)
#  phase : start development
#  date  : 2002-10-25
#######################################
  ## 2nd stratum reads input file - no
  ## processing
  my $line="";
  while ($line=<STDIN>){
    ### 3rd stratum reads only 'sections'
    ### i.e., it skips over front lines
    ### that do not start with numbers
    ### below, it detects the start of the
    ### first section
      if  ($line =~ /^[0-9]+\)/){ last; }
  }
    ### read the rest of the input log file
    ### for processing
        ##### stratum 5
        ##### define the flags neccessary for
        ##### 'field' detection
        my $fldText;
        my $textDetected;
        my $fldStartTime;
        my $fldStopTime ;
    do {
      #### we start generating output (a
      #### simple copy)
      if ($line =~ /^[0-9]+\)/ ) {
      #### here we are at the beginning of
      #### a new section
      #### that also means the previous
        print "---------------------\n";
        ##### print only desired lines
        print$fldText.$fldStartTime.$fldStopTime
        if ($textDetected);
        ##### initialize/reset 'field' detection
        ##### flags
        $fldText =""; $textDetected=0;
        $fldStartTime ="";
        $fldStopTime  ="";
      } else {
      #### here we are inside a section
        ##### detect 'fields' and set flags
        if ( $line =~ /^  Text     :(.+)/)  {
        $fldText  =$1."\n"; $textDetected=1;}
        if ( $line =~ /^  Start Time:(.+)/) {
        $fldStartTime =$1."\n"; }
        if ( $line =~ /^  Stop Time:(.+)/)  {
        $fldStopTime  =$1."\n"; }
      }
        $line=""; #### Perl does not accept
        ##### break loop 'next' in this context
      print $line;
    } while ($line=<STDIN>);
```

**Fig. 7 LFAP: Final Form with Five Strata**

First, the paradigm allows the creation of programs with *adjustable functionality*. In the example presented, the layers were fine-tuned in a way that meant successive reductions of the output's size, towards reaching a selection goal. In other applications, the opposite can happen: deeper the stratum, more the details provided in the program's output. Yet in other applications we can image strata having quasi-independent contents, which implies that lower level strata would override part of the higher-level strata's functionality.

Second, a powerful tool for *flexible, rapid software construction* is presented to the developer. SP, similar to new approaches for efficient software development such as *extreme programming* (XP) [11], *agile development methods* [13], or *rapid application development* [14], focuses on rapid production of quality software. We see SP as a new technique for evolutionary development, including exploratory development and rapid prototyping. The benefits of these types of development, in particular the early creation of programs that are gradually extended and enhanced based on users' feedback, are well documented in the software engineering literature [1, 12].

Third, being a new approach that offers a new software construction paradigm, SP brings a number of *new research and development challenges*. Some details on these are presented in the next section, which outlines specific topics we intend to explore in the near future. In general terms, however, it is clear that from a coding-focused technique, SP has the potential to spread across the entire software life-cycle. The example presented constitutes an "on-the-fly," rapid unit (or detailed) design as well as a basis for unit testing. Evidently, SP's principles can be applied to the earlier stages of the software development process: high-level designs can be created in terms of strata-based software and, continuing backwards our "walk" through the phases of software life-cycle, the specification of the software can be guided by the program strata (software strata) concepts. But it is not only towards the earlier phases of the software life-cycle where SP can be extended; evidently, SP principles and techniques could affect later phases as well. It is reasonable to envision that software specified, designed, and coded according SP principles will also be integrated, tested, and maintained in accordance with these principles. Hence, we believe, the SP paradigm can extend over the entire scope of the software life-cycle.

While some of SP's benefits are mentioned above, in particular flexibility and efficiency, the new approach also brings a number of challenges. At this point in time, we see as the main challenge for SP the requirement for a *mindset shift* entailed by the new paradigm: both developers and users will need to adapt to the new paradigm and see software as a set of strata, each stratum defined by specific characteristics and specific functionality. All other challenges that we, at present, are aware of are related to this major challenge and can be classified broadly in two categories: *supporting methodology* and *supporting tools*. We intend to approach both these areas, as described next.

## 4 FURTHER WORK

Our future work on SP will encompass both the definition of a systematic SP methodology and the design of a set of assisting tools. In parallel, we intend to extend the application of the proposed approach to new, more complex case studies. The next step will be to apply the SP approach to module design and integration. The same database management application from which we have taken the LFAP example to illustrate unit development will be used in a larger study to exemplify higher level design and module integration and testing.

At this time, we think that we need first the support of certain *SP tools*, which will reduce the time needed for applications. We have already started working on such tools and believe that both exercising SP on new software development cases and designing supporting tools will provide us with valuable feedback for necessary methodological adjustments. Regarding tool support for SP, specific tools that we are currently working on are:

- An *SP code editor*, which will allow easy demarcation and manipulation of strata. In essence, the code editor is intended to provide: i) automated indentation of strata; ii) switching mechanisms for on-screen presentation of program strata; and iii) a "layered" interface with the compiler;
- An *SP flowchart editor*, that will provide to unit and module design a similar type of support the code editor provides to coding;
- A *specialized SP Perl-based language* (in the first instance), which will include simple extensions for development support at the programming language level.

From a methodological point of view, we have two major objectives. First, we aim to define a *systematic procedural framework for strata-based software construction* in terms of steps, deliverables, and guidelines for applying the SP principles across the software life-cycle, from specification, through design, implementation, integration, and testing to maintenance/evolution. Second, we intend to develop a *formalism* for software strata and strata-based software construction that would allow us to reach the more demanding areas of safety- and security-critical systems. We also consider other research directions, in particular the application of program strata to object-oriented development.

## 5 CONCLUSIONS

We have introduced in this paper a new software construction approach, denoted *stratified programming* (SP). The concepts on which SP is based, *program stratum* and *program execution layer* were also described, and the approach itself was illustrated with a simplified, yet detailed example. A discussion on the potential benefits of the new approach was also included and the main challenges raised by SP were overviewed. Potential extension of SP to the more general approach of *strata-based software construction* (SSC) was also indicated and a number of research and development topics, including specialized methods and tool support, were described.

We believe that building and executing programs using strata not only proposes a novel, innovative software engineering approach particularly suitable to rapid and efficient construction of flexible software, but also provides the seeds of several very interesting and challenging research and development topics pertaining to the software engineering spectrum.

## REFERENCES


[ 1] Ghezzi, C., Jazayeri, M., and Mandrioli, D., *Fundamentals of Software Engineering*, 2nd Ed., Prentice-Hall, 2002.

[ 2] Meyer, B. "Software Engineering in the Academy," *IEEE Computer*, 34 (5), p. 28-35, May 2001.

[ 3] Wirth, N., "Program Development by Stepwise Refinement," *Comm. of the ACM*, 14 (4), pp. 221-227, April 1971.

[ 4] Shalloway, A., and Trott, J. R., Design *Patterns Explained: A New Perspective on Object-Oriented Design*, Addison-Wesley, 2001.

[ 5] Fidge, C., Kearney, P., and Utting, M., "A Formal Method for Building Concurrent Real-Time Software," *IEEE Software* 14 (2), pp. 99-106, 1997.

[ 6] The XML Epic Editor, Arbortext web-site, http://www.arbortext.com/html/ee_close-up.html, accessed Nov. 14, 2002.

[ 7] CVS, Concurrent Versions Systems web-site, http://www.cvshome.org/docs/manual/cvs.html, accessed Nov. 14, 2002.

[ 8] Microsoft Visual Source Safe web-site, series of technical articles, accessed Nov. 14, 2002, msdn.microsoft.com/ssafe/technical/articles.asp,

[ 9] Dascalu S., and Hitchcock, P., "Harmony: An Environment for the Combined Use of UML and Z++ in Software Specification," in J. Parsons and O. Sheng (editors), *Procs. of the 11th Workshop on Information Technologies and Systems*, New Orleans, LA, pp. 103-108, Dec. 2001.

[10] Cavalcanti, A., and Naumann, D., "A Weakest Precondition Semantics for Refinement of Object-Oriented Programs," *IEEE Trans. on Software Engineering* 26 (8), pp. 713-728, Aug. 2000.

[11] Beck, K., *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.

[12] Pressman, R., *Software Engineering: A Practitioner's Approach*, 5th Ed., McGraw-Hill, 2001.

[13] Highsmith, J., and Cockburn, A., "Agile Software Development: The Business of Innovation," *IEEE Computer* 34 (9), pp. 120-122, Sep. 2001.

[14] McConnell, S., *Rapid Development*, Microsoft Press, 1996.