STRATIFIED PROGRAMMING INTEGRATED DEVELOPMENT ENVIRONMENT (SPIDER)

Sergiu Dascalu*, Adrian Pasculescu, Josh Woolever*, Eric Fritzinger*, Vivek Sharan***

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

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

Abstract

This paper describes the functionality required for a development environment that supports stratified programming (SP), a novel software development method that we have proposed recently [1, 2]. In this paper we discuss the case when program strata are controlled outside the programming language and present the main features of the SPIDER environment for strata creation and SP program execution. The central part of the development environment is a source code editor, whose specific strata manipulation functions are described in detail in the paper. An example of stratified XSL/XSLT code is also included to illustrate the main concepts of stratified programming.

Keywords: program stratum, layered functionality, stratified programming, integrated development environment, source code editor.

1 INTRODUCTION

Abstraction is important at all levels of knowledge manipulation. In problem solving, abstraction is typically accompanied by refinement, their combination providing a powerful means for dealing with complexity. In software development, abstraction and refinement (iterations) are omnipresent, incorporated in practically all software development techniques and tools (e.g., [3, 4, 5]). One such technique is the incremental software development [6], whose principles are included in our proposed *stratified programming* (SP) approach.

As described in [1], what SP proposes is building and executing programs using a strata structure which, we believe, allows for both adjustable program design *and* adjustable program execution. This supports rapid and efficient development of *flexible programs* with *layered functionality* in which strata (program layers) can be easily created and manipulated. This is a novel approach, since typical incremental or program refinement approaches [e.g., 7] successively increase the detail level of a program specification, possibly up to code generation. In SP, in addition to incremental program creation, we propose maintaining the correspondence between design strata and execution strata, the latter being possibly selected by a user during program operation. From both development (including specification, design, coding, and testing) and execution points of view an SP program can be seen as having different *shapes*, each determined by the specific strata configuration considered in a given context.

Being only recently proposed, SP needs both theoretical elaboration and, essential from a practical point of view, appropriate tool support [1]. In order to supply the latter we have started work on a supporting environment, tentatively entitled SPIDER (an acronym from **S**tratified **P**rogramming **I**ntegrated **D**evelopment **E**nvi**R**onment). This environment and its main component, the SP code editor, are presented in this paper. A brief discussion of the extension of SP to the more general approach we denoted *strata-based software construction* (SSC) [2] is also included.

The paper, in its remaining part, is organized as follows: Section 2 reviews the main concepts of stratified programming, Section 3 presents a short example of SP code, Section 4 describes the design principles of the SPIDER environment, and Section 5 concludes the paper with notes on SP-related directions of research and development.

2 STRATIFIED PROGRAMMING CONCEPTS

In [1] we have introduced the main SP concepts and in [2] we have suggested the extension of SP (focused on unit design and implementation) to SSC (aimed to cover other phases of the software process, in particular specification and high-level design). Here, we refine the definitions presented in [1] and introduce several additional terms.

There are several possible definitions for program strata. The one that we use in this paper describes a *program stratum* as a portion of a program module that adds a contribution to the functionality of the module provided by the previous, higher-level strata. For example, a new stratum added to an existing program module's strata structure (see *program shape* or *program configuration* below) uses a different subspace of the module's input space and/or refines the program's output space.

Each stratum has a property denoted *stratum depth*, which naturally represents the stratum's position within the program's strata structure. Stratum depth is similar to the *isobath* parameter in ocean cartography [8] and is denoted by a natural number, starting from 1. More precisely, the "surface stratum" of a program module has depth 1, the next stratum has depth 2, and so forth.

For a given program module, each subset of strata corresponds to a strata structure denoted *program shape* (or *program configuration*). For instance, strata 1 and 2 make up program shape 2 (denoted *program_name_s2*), strata 1, 2, and 3 constitute program shape 3 (denoted *program_name_s3*), and so forth.

Regarding strata configuration, for non-procedural programming languages such as XSLT [9], it is possible to consider arbitrary combinations of strata such as 1, 3, and 6, which lead to a program shape with the postfix s_1+3+6. However, in practice this alternative involves more complex strata elaboration and manipulation, and hence we defer for the moment its discussion.

As regards strata creation, we suggest using a structuring criterion based on the "expansion/contraction" of the program module's *input* and *output spaces.* For example, in the case of a program in which each additional stratum restricts the type of files admitted for input (e.g., from any file, to a text file, to a log database file, etc.) we can speak of a *contraction* of the input space along the depth coordinate of the program. (To be more precise, the input space for the program remains the same; what changes is the *input subspace* used in a given stratum.) Similarly, we can imagine *expansion* of the input space as well as contraction and expansion of the output space. Consequently, in principle strata definition can follow four approaches, resulting from the possible combinations of input and output space evolutions (contraction/expansion). Certainly, the I/O criterion for strata definition is not the only one that can be used when structuring a program module in strata; other criteria such as the set of services provided can also be applied.

From a visual representation point of view —very important when designing SP supporting tools— the level of code indentation is used to demarcate strata. Within each stratum traditional code indentation can be used in a limited way, but for the sake of simplicity we omit this detail in the present paper.

Before introducing a short example of SP code, it is worth noting that in this paper we approach SP from a language-independent point of view, a point of view in which strata are defined at a conceptual level, "outside" the programming language. For example, strata are not delimited by internal constructs such as a begin_stratum or end_stratum, and language constructs such as refresh (V,N) (which allows variable V on stratum M to regain the value it held on stratum N) are not included. Specialized SP versions of programming languages constitute the subject of future research.

3 A BRIEF EXAMPLE

In order to illustrate the main SP concepts, let us consider a simple module, denoted Config (Figure 1). In this module an XML configuration file from a web service for web page changes detection (diffweb) is updated [10]. Note that from a functional point a view, this module has a very thin granularity of strata. In practice, however, "thicker strata" (that is, with more functionality incorporated) should normally be created.

```
<!—start of level 1 stratum (outermost)-->
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl=
 "http://www.w3.org/1999/XSL/Transform" >

<!-- Copyright 2000-2003 ALPAS Solutions -->
     <!—start of level 6 stratum -->
     <xsl:param name="itemNumber"/>


      <!—start of level 7 stratum -->
      <xsl:param name="today"/>

 <!—start of level 2 stratum -->
 <xsl:template match="*">

  <!—start of level 3 stratum -->
  <xsl:copy >

   <!—level 4 stratum -->
   <xsl:for-each select="@*">

    <!—level 5 stratum -->
    <xsl:choose>

     <!—continuation of level 6 -->
     <xsl:when test=
        "../@number!=$itemNumber">
     <xsl:copy/></xsl:when>
     <xsl:otherwise>

      <!—continuation of level 7 -->
      <xsl:attribute name="diffDate">
      <xsl:value-of select="$today"/>
      </xsl:attribute>
     </xsl:otherwise>
    </xsl:choose>
   </xsl:for-each>
   <xsl:apply-templates/>
  </xsl:copy>
 </xsl:template>

</xsl:stylesheet>
```

Fig. 1 Config: Example of SP code in XLST

In the Config example shown above:

- Stratum 1 (program shape 1, denoted Config_s1) defines the style sheet copyright, date and comments. It simply creates a style sheet transformation that works with any XML document and extracts all text nodes from the XML input document.
- Strata 1 and 2 (which make up program shape 2, Config_s2) define a transformation that generates an empty XML document.
- Strata 1, 2 and 3 (Config_s3) define a transformation that extracts only the root node of the input XML document.
- Strata 1, 2, 3 and 4 (Config_s4) extract all nodes, regardless of their attributes.
- Strata 1, 2, 3, 4, 5 and 6 (Config_s6) extract all nodes and attributes for nodes that do not have a specified value of the @number attribute.
- Strata 1, 2, 3, 4, 5, 6 and 7 (Config_s7) additionally replace the values of the attributes of nodes that do have a specified value of the @number attribute.

With the main SP concepts introduced, the characteristics of the supporting integrated development environment for our proposed technique are discussed next.

4 DESIGN PRINCIPLES FOR SPIDER

The core of the SPIDER environment is, from our perspective, the editor. For this reason, we focus in this paper on the editor and only briefly mention other features that need to be included in the environment. In particular, besides the editor, SPIDER will incorporate traditional IDE features that allow program compilation, debugging, and execution. Additional facilities such as multiple windows, help system, and add-on tools will also be considered.

4.1 Lax vs. Rigid Stratification

In practice, a developer might use one of the following two versions of stratifications, *lax stratification* or *rigid stratification*, as well as a combination of both. For *lax stratification* the strata are not nested from the point of view of the depth. This means that a stratum may be followed immediately by a stratum at a much deeper level. For instance, in Figure 2 there is a "jump" from stratum 2 to stratum 5. In the case of *rigid stratification* (Figure 3) all levels are nested and "jumps" possible in lax stratification are not allowed. This implies that a tree representation of the code can be used. In practice we can enforce rigidity using comments, and considering comments as "neutral code" blocks on intermediary ("jumped-over") strata. In Figures 2 and 3 transitions from a stratum to another are represented with dashed arrows.

Based on our experience, we notice that stratification can be applied successfully not only for modules written in functional (non-procedural) languages such as SQL or XSLT but also —and especially— for programs written in assembly languages (e.g., code for microcontrollers used in robot control), C, Pascal, and so forth.

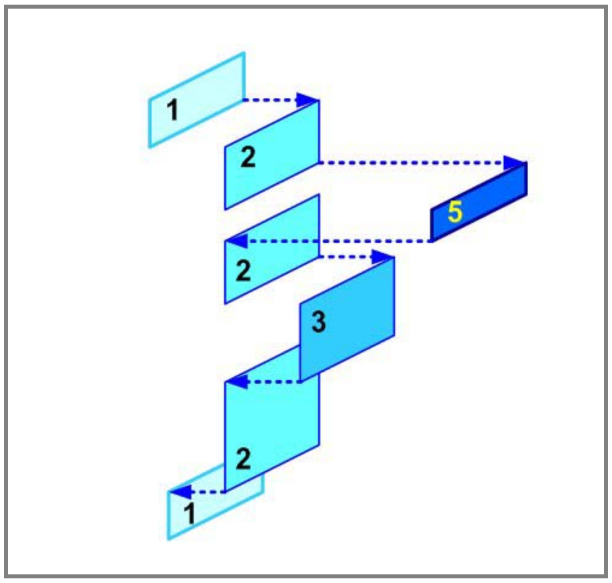


Fig. 2 Lax stratification

As stated above, combination of approaches can be used when developing a given program module. For example in Figure 1, lines 6-12 are written following a lax stratification approach, while lines 13-41 are created in a "rigid" way.

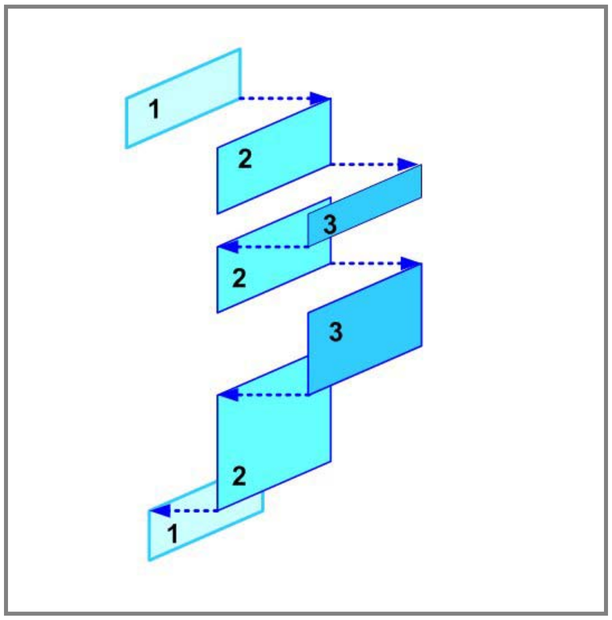


Fig. 3 Rigid stratification

4.2 The Editor

As indicated in section 2, code indentation provides the easiest discrimination of strata. Starting from this observation, we focus next on facilities needed in the source code editor that will be part of SPIDER.

The editor should have at least the following initial set of strata manipulation commands:

Presentation commands

- *Reveal next stratum*. Show the next stratum present in the module's code, but not visible on the screen.
- *Hide deepest stratum*. For example, hide code lines 10, 11, and 30-33 (stratum 7) when the image of the module shown on the screen is that presented in Fig. 1.
- *Show to cursor*. Show strata from the outermost stratum down to the stratum where the cursor is currently positioned.
- *Show from cursor*. Show strata down from the cursor, i.e., the stratum on which the cursor is positioned as well as all the deeper strata.
- *Select stratum*. Show the stratum where the cursor is positioned at a given moment during the code editing.

The above presentation commands should be available in SPIDER through menu items as well as through toolbar icons. Toolbar icons corresponding to the five presentation commands detailed above are shown, in order, in Figure 4.

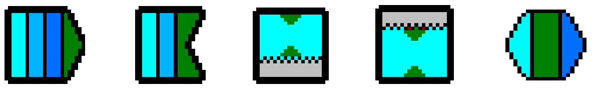


Fig. 4 SPIDER-specific toolbar icons

Navigation commands

Vertical arrows will move the cursor inside the visible code. For example, if a stratum is hidden, the vertical movements will stay only on the visible strata. The developer must issue one or more *reveal next stratum* commands in order to navigate on hidden strata.

Line numbering can also give an indication of the hidden strata. All the lines of the code will be numbered regardless of the visibility. Optionally, there might be an additional column that gives the depth number of each stratum.

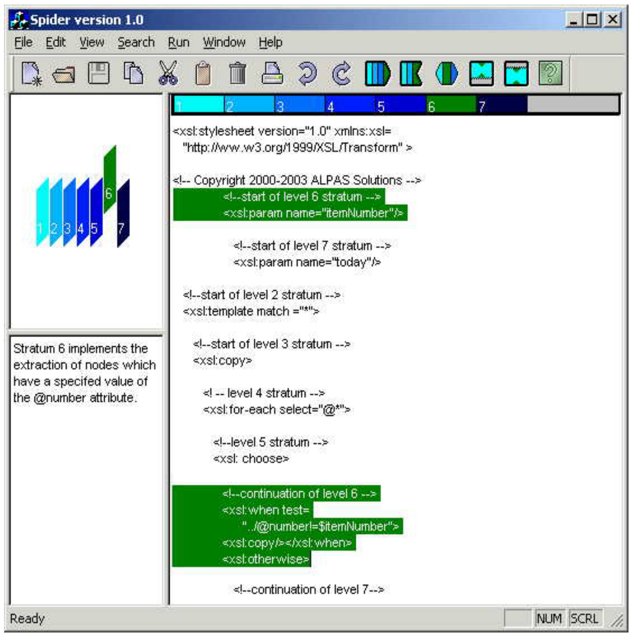
We should also have some other way of marking of the hidden strata. For example one might use "29..." following the line numbers in lines that precede a hidden stratum and "…34" for lines that follow immediately after a hidden stratum.

Other SPIDER commands

- *Save down to the deepest presented level*. This operation should prompt the user with a specific message if discarding changes in lower levels is involved. For instance, Config_s5.xsl could be the saved version of the Config program when only the first five strata of the program are shown.
- *Compile/run/debug only the presented code*. Compile, debug, or execute only the source code presented on the screen. This will ignore the hidden code.
- *Search within visible code*. Search only in the visible portion of the program; this is in addition to the default search on the entire code which requires the presentation of the program's maximal shape (all strata).
- *Replace within the visible code*. Similar to the "search within visible code" command described above.
- *Mark areas* and *select areas* of the visible code.

Visual strata presentation enhancements

The backgrounds of program strata could be presented in different colors, for example using various shades of blue, from light to dark, similar to the representation of an ocean's depth in marine maps. In fact, using this metaphor, an "ocean bar" which depicts the program's strata organization and indicates the selected stratum will be attached at the top of the right pane of the SPIDER window. The main SPIDER window is presented in Figure 5, while presentation details based on the "ocean" metaphor are provided in Figures 6 and 7.



```
<xsl:stylesheet version="1.0" xmlns:xsl=
  "http://www.w3.org/1999/XSL/Transform" >

<!-- Copyright 2000-2003 ALPAS Solutions -->
          <!--start of level 6 stratum -->
          <xsl:param name="itemNumber"/>

            <!--start of level 7 stratum -->
            <xsl:param name="today"/>

  <!--start of level 2 stratum -->
  <xsl:template match ="*">

    <!--start of level 3 stratum -->
    <xsl:copy>

      <! -- level 4 stratum -->
      <xsl:for-each select="@*">

        <!--level 5 stratum -->
        <xsl: choose>

          <!--continuation of level 6 -->
          <xsl:when test=
              "../@number!=$itemNumber">
          <xsl:copy/></xsl:when>
          <xsl:otherwise>

            <!--continuation of level 7-->
```

Stratum 6 implements the extraction of nodes which have a specifed value of the @number attribute.

Fig. 5 SPIDER's main window

Regarding the contents of the SPIDER main window shown in Figure 5, we note that in the window's left pane program strata are shown in the top compartment in a graphical form that allows direct manipulation, in particular stratum selection (Figure 6). Based on this initial "cartographic representation" of the code various strata manipulation features can be designed. In the lower compartment of the main window's pane a brief description of the selected stratum can be incorporated to remind the developer of the criteria used for that stratum's definition. Finally, in the right pane of the SPIDER window the selected stratum is highlighted. For example, in Figure 5 the selected text belongs to stratum 6, and this fact is indicated using the same color (teal) in the "ocean bar" shown in Figure 7.

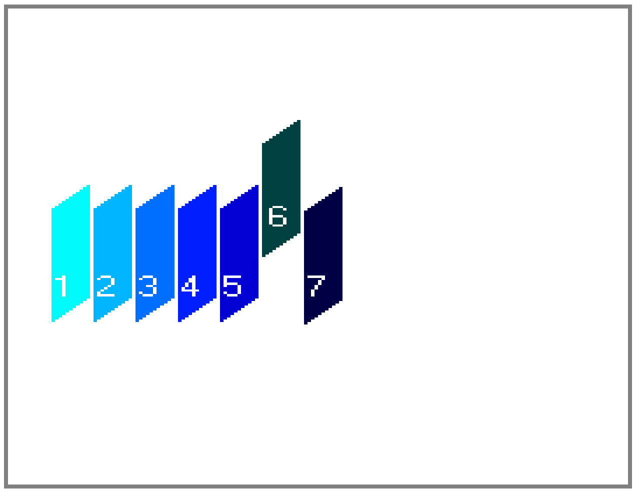


Fig. 6 SPIDER left upper pane

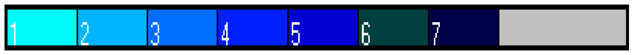


Fig. 7 SPIDER "ocean bar" (top of right pane)

Internal structures

One distinction between usual text (code) editors and stratified editors is that moving vertically might involve jumping over several hidden lines of code. The position of the cursor is important for inserting new text. One possible implementation will keep two temporary areas and a number of indicators (flag). For example:

- Max_depth_visibility := a number equal to the number of visible strata.
- Upper_file_object := the file content up to the current position of the cursor, including the line where the cursor is currently located.
- Lower_file_object := the content of the source file, below the cursor's current location.
- Visibility_strata_set := array with the indexes of visible strata.

The above objects should maintain information about each line (for example the current depth). Once the cursor is moved up, portions of the Upper_file_object are move to the Lower_file_object. Evidently, these are only small parts of an implementation solution that is expected to be significantly more complex than that of a regular source code editor. This is, in fact, the price we are willing to pay in order to support a software development approach that promises not only significant increase in the programmer's productivity but also greater flexibility in the structuring *and* execution of programs.

5 CONCLUSIONS

SP is a new software development approach that we have recently proposed. Starting from unit design, implementation, and coding, we envisage extending SP to other phases of the software process, in particular to specification and higher-level design. SP and its extended version SSC provide the premises of several very interesting research and development directions in the software engineering domain. We have indicated elsewhere [1, 2] that the main challenge of the new approach is the requirement for a mindset shift (paradigm shift in software construction) for both software developers and software users. Both developers and users have to learn to "think in strata" and see programs organized in layers of functionality built according to various structuring criteria.

In terms of needed developments, SP requires both methodological refinement and tool support. To address the latter, we have presented in this paper the design of the SPIDER environment which, when fully functional, will allow us to experiment with the SP approach and will provide valuable feedback for necessary methodological adjustments. In parallel, we have started work on the theoretical foundation of the approach, in particular on the formalization of strata and on the definition of a set of SP-specific programming constructs.

Work on SPIDER's implementation has also been started in the Computer Science Department at the University of Nevada, Reno, in collaboration with Alpas Solutions, Toronto.

REFERENCES

[1] A. Pasculescu and S. Dascalu, "Stratified Programming: Towards a New Paradigm for Software Development," *Proceedings of the 18th Intl. Conference on Computers and Their Applications (CATA-2003)*, Honolulu, Hawaii, pp. 263-268, 2003.

[2] A. Pasculescu and S. Dascalu, "Strata-based Software Construction," accepted at the *7th Intl. Conference on Systemics, Cybernetics, and Informatics (SCI'2003)*, Orlando, Florida, July 2003.

[3] D. Sotirovski, “Heuristics for Iterative Software Development,” *IEEE Software*, 18(5): 66-73, 2001.

[4] P. Koppol, R.H. Carver, and T. Kuo-Chang, “Incremental Integration Testing of Concurrent Programs,” *IEEE Transactions on Software Engineering*, 28(6): 607-623, 2002.

[5] XML Epic Editor, Arbortext web-site, accessed February 10, 2003, http://www.arbortext.com/html/ ee_close-up.html

[6] N. Wirth, “Program Development by Stepwise Refinement,” *Communication of the ACM*, 14 (4), pp. 221-227, 1971.

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

[8] Forthright's Phrontistery web site, maintained by S. Chrisomalis and J. Chrisomalis, *Isolines* page, accessed February 12, 2003, http://phrontistery.50megs.com/ contour.html

[9] E.M. Burke, *Java and XLST*, O’Reilly and Associates, Inc., 2001.

[10] Diffweb web site, Alpas Solutions, accessed February 12, 2003, http://www.diffweb.com