Generating Consistent Program Tutorials

Thomas Vestdam

Department of Computer Science
Aalborg University
Fredrik Bajers Vej 7E
DK-9220 Aalborg
Denmark
E-mail: odin@cs.auc.dk

Abstract. In this paper we present a tool that supports construction of program tutorials. A program tutorial provides the reader with an understanding of an example program by interleaving fragments of source code and explaining text. An example program can for example illustrate how to use a library or a framework. We present a means for specifying the fragments of a program that are to be in-lined in the tutorial text. These in-line fragments are defined by addressing named syntactical elements, such as classes and methods, but it is also possible to address individual code lines by labeling them with source markers. The tool helps ensuring consistency between program tutorial and example programs by extracting fragments of source code based on the fragment specifications and by detecting when a program tutorial is addressing program fragments that do not exist. The program tutorials are presented as online resources, providing navigation to other documents, such as reference manuals, other tutorials, internal documentation, and also to a presentation of the source code fragment in its original context. We have produced three example program tutorials in order to get some experience with the tool, and we see potential in using the tool to produce program tutorials to be used for frameworks, libraries, and in educational contexts.

1 Introduction

Tutorials offer a pedagogical set of guidelines that provide the tutorial reader with a simplified learning space. Such guidelines, or instructions, can for example guide the reader through some important functionality of an application. As stated in [6]:

"Tutorials … provide(s) a supportive and safe educational environment for learning processes or products. They usually include interactivity, and the content and pace are controlled, to some degree, by users. Tutorials have a broader pedagogical scope than both documentation and help."

The reader can set the pace by skipping through sections of the tutorial. Hence, the reader can go right to the point where the needed information is, and *get started fast* [5]. In addition, online tutorials can include links to other types of information such as manuals and in-depth explanations of specific concepts.

A major advantage of using tutorials is the option of getting started fast. Carroll [5] found that people prefer to interact before they learn, for example before reading a manual. This is also a concern in [4], where Østerbye realizes that tutorials are important due to Carroll's "paradox of sense-making":

"To learn, they (the users) must interact meaningfully with the system, but to interact with the system, they must first learn." [5]

Tutorials can in general be useful in many contexts - whenever a reader needs an introduction to a product, concept, or process. In this paper, we will retain to a specific type of tutorials called *program tutorials*.

We define a program tutorial as a document that informs the user about the internal properties of a program, especially with the goal of using the program for construction or composition of another program.

Program tutorials are very useful as end-user documentation of libraries and frameworks [19], where the end-user is a programmer using a framework or library. In addition, tutorials are also often used in educational contexts where students are exposed to real programs when introduced to programming language basics, or exemplar solutions to common problems. Common for all these tutorials is that they present *example programs*.

Example programs are very useful for two reasons (at least). First, studies have indicated that people learn by induction from concrete examples, rather than being told how to do things [5]. Secondly, having example programs that are ready to be compiled and executed gets the reader started fast [4].

As a program, library, or framework evolves, interfaces will change. Hence, example programs must change accordingly. When an example program is changed so must any tutorial having parts of the example program interwoven in the tutorial text. This can make maintenance of tutorials a difficult task, especially if example programs are manually pasted into tutorials.

In a literate program [1] documentation and source code are interleaved in arbitrary sized chunks, chosen by the developer. Literate programs can therefore serve as tutorials. Furthermore, Literate Programming facilitates keeping example programs and tutorial text in accord because documentation and source code are in close physical proximity.

In [4], Østerbye presents an approach utilizing Literate Programming for creating program tutorials. The tool *noweb* [12] is used to produce a program tutorial - The BetaSIM tutorial. The tutorial is made up by a single scenario, which is elaborated into an increasingly complicated synchronization scenario. The tutorial source contains both example program chunks and tutorial text tagged in noweb syntax. The example programs are extracted from tutorial source by “tangling” the tutorial source whereas the final online tutorial is “woven” from the tutorial source.

One serious problem was encountered in this approach. It was found necessary to be able to extract several different programs from the tutorial (i.e. having several variations). However, this is not possible when using noweb. Consequently, it was necessary to state each variant of the example programs in full. It is concluded that it would be attractive to have some kind of specialized literate tool that can handle variations.

Another drawback of using Literate Programming is that it is not possible to use the fragments of an example program in more than one context. This is for example useful when having tutorials for different audiences that address the same example programs. In addition, in order to produce literate programs programmers need to reorganize and split apart their example programs.

In this paper, we propose using Elucidative Programming [2][3][8] to accommodate such needs. Elucidative Programming is a variation of Literate Programming. In Elucidative Programming program source files and documentation live as two separate entities tied together through mutual navigation in a two-frame online layout (see figure 1). The program units that are addressable from the documentation are the program abstractions (syntactical elements such as classes, methods, and variables), or special comments in the program working as anchors - called *source markers*. The documentation tool, called an *elucidator*, uses language knowledge in order to identify these abstractions in the program source files. This enables the elucidator to provide two-way navigation between anchors in the documentation and their respective destination in the program source files. In the presentation of program source files, program abstractions are rendered as links providing navigation from a program abstraction to any part of the documentation addressing that specific program abstraction. In addition, hyperlink-navigation is provided between identifies and their definition.

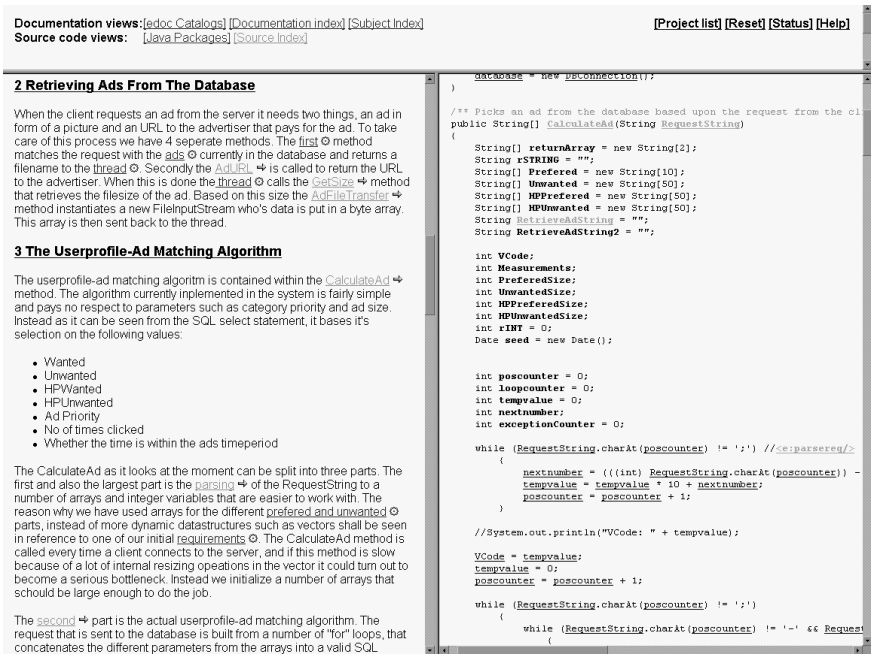


Fig. 1. An example of the presentation of an elucidative program in a web-browser. The top frame contains various navigational features, such as an index of program abstractions and a documentation index. The left frame presents documentation and the right frame presents program files (i.e. source code).

In Elucidative Programming a simple documentation language is used to markup text such as sections and subsections. The language also enables the programmer to specify links going from documentation to source code. Besides a destination, a link can be given a type, for example denoting a link as a strong or weak reference. This enables the documentation reader to perform selective navigation.

In this paper, we will present a tool that supports programmers when writing tutorials. The tool support is implemented as an extension of the *Java Elucidator* [8], and allows fragments of example programs to be addressed from the tutorial source. When the tutorial is presented in a web-browser, the respective program fragments are extracted from the source and in-lined in the tutorial. The tutorials that are produced using the tool are online resources, providing the reader with a number of navigational features when viewed in an ordinary web-browser.

We contribute by providing an addressing mechanism and a simple extraction language that allows a tutorial author to specify fragments of a program. These fragments are extracted from the program each time the program tutorial is presented. This relieves the tutorial writer from manual synchronization of example program source and tutorial source.

We also contribute by supplying tool support that can report when a tutorial is addressing program fragments, such as a method or class, which no longer exists or has changed signature. However, it is not all types of change that can be detected, but as we shall see, by using source markers more subtle changes in the source code can be detected.

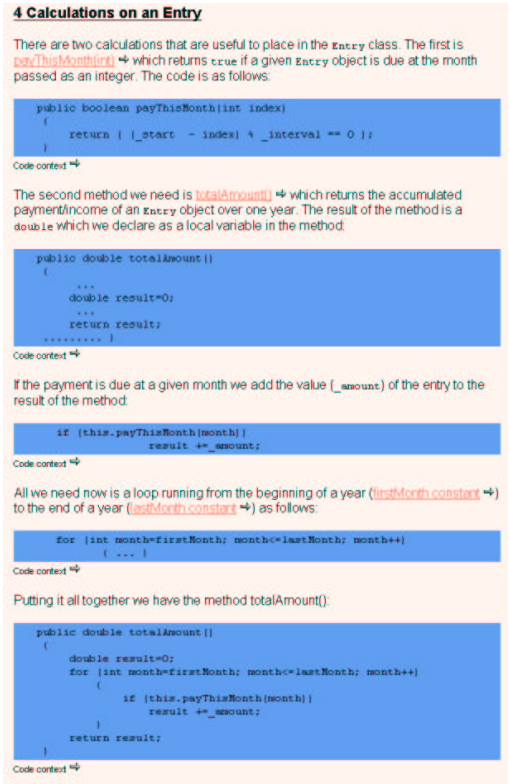
In addition we contribute by adding to the idea of Elucidative Programming by providing the option of having source code in-lined in documentation. This approach is an attractive alternative to Literate Programming, as the programmer is relieved from having to chunk example programs when writing program tutorials.

The rest of this paper is organized as follows. In section 2 we will take a closer look at what a program tutorial is and the problems connected with keeping a program tutorial consistent with a library, framework, or program. In section 3, we will present a documentation language

extension, for the Java Elucidator, which allows the programmer to specify fragments of a program that are to be *extracted* and *in-lined* in the documentation – hereby supporting creation of program tutorials. In section 4, we explain the tutorial tool in more detail, including how the source code fragments are extracted and how program tutorials are presented in a web-browser. In section 5, we will discuss the current tool, by looking at the strengths, weaknesses and potential applications that we see. Finally, in section 6 we will conclude on the current work.

2 Program Tutorials

In this section we will take a look at what a typical program tutorial is in order to outline the problems that tool for creating such tutorials should aim to relive. A typical program tutorial is illustrated in figure 2. As can be seen the program fragments from the example program are woven into text explaining the different program fragments. The example program is a suggested solution to a recent student assignment at our department [11].



4 Calculations on an Entry

There are two calculations that are useful to place in the `Entry` class. The first is payThisMonth(int) ➪ which returns `true` if a given `Entry` object is due at the month passed as an integer. The code is as follows:

```
public boolean payThisMonth(int index)
{
    return ( (_start  - index) % _interval == 0 );
}
```

Code context ➪

The second method we need is totalAmount() ➪ which returns the accumulated payment/income of an `Entry` object over one year. The result of the method is a `double` which we declare as a local variable in the method:

```
public double totalAmount()
{
    ...
    double result=0;
    ...
    return result;
..... }
```

Code context ➪

If the payment is due at a given month we add the value (`_amount`) of the entry to the result of the method:

```
    if (this.payThisMonth(month))
                result += _amount;
```

Code context ➪

All we need now is a loop running from the beginning of a year (firstMonth constant ➪) to the end of a year (lastMonth constant ➪) as follows:

```
    for (int month=firstMonth; month<=lastMonth; month++)
        { ... }
```

Code context ➪

Putting it all together we have the method totalAmount():

```
public double totalAmount()
{
    double result=0;
    for (int month=firstMonth; month<=lastMonth; month++)
        {
            if (this.payThisMonth(month))
                result += _amount;
        }
    return result;
}
```

Code context ➪

Fig. 2. A fragment of an elucidative tutorial. The section "Calculations on an Entry" explains two simple methods. The first method is present in its entirety, whereas the second is presented in four steps.

An example program is related to what we will call a *surrounding world.* The surrounding world is for example a library, framework, or a topic in an educational context. As a consequence of the "paradox of sense-making", example programs that can be compiled and executed should always accompany tutorials. This means that the tutorial reader can start interacting with the example programs as well as the world surrounding the tutorial.

The tutorial itself is used as a guideline bringing the reader deeper and deeper into the details of an example program. In turn the reader gains an understanding of how to use a given

library or framework, or gives the reader an understanding of a topic of concern in an educational context.

A tutorial can be seen as a thread of understanding going from high-level explanations to low-level explanations. By having fragments of program examples in-lined in the tutorial text the reader can concentrate on a fixed thread of understanding [9]. However, as argued in [7] the reader sometimes needs to leave the thread in order to gain additional knowledge. For example, tutorials in the Java Trail [10] often include links guiding the reader to relevant places in the Java API, Java Language specification, other tutorials, or other relevant external documents.

Another relevant example is the BetaSIM tutorial [4]. BetaSIM is a framework for discrete event simulation in the programming language BETA. The BetaSIM tutorial is an online resource explaining a number of example programs that takes the reader through the basics of the framework. The tutorial also links all usage of framework classes and methods to a traditional reference-manual, providing the curious reader with more information.

As defined in section 1, a tutorial informs the user about the internal properties of a program. This is often done in an *incremental* manner, going from the basics to more complicated matters. This is illustrated in figure 2, where a method called `totalAmount()` is explained by first introducing a local variable, then explaining when to increment the variable, then explaining how this is done in a for-loop, and finally revealing the entire method. In program tutorials we often wish to present fragments of the example program in such a manner. The tutorial writer chooses the actual order in which fragments of an example program are presented. The example in figure 2 cannot be re-produced using Literate Programming because it contains *repetitions* of source code. Using Literate Programming it would be necessary to introduce chunks, and the final presentation of the entire `totalAmount()` method would consist of four references to chunks. We hypothesize that a program tutorial that contain repetition of source code rather than references to chunks has advantages in terms of flow of reading.

Furthermore, it is some times necessary to present example programs in different *variations*. For example, first a simple example program is explained. Then more complexity is introduced and the program evolves into a new example program. We distinguish between two types of variations:

Simple variations: These are variations that can be produced by presenting fragments of the example program in different order and detail. For example, parts of the example program are extended, as in the tutorial fragment in figure 2.

Complex variations: These are variations where parts of the example program are redefined. Each complex variation must be an independent example program because a given variation cannot be directly extracted from any of the other variations. For example, in figure 2, if the method `totalAmount()` is later redefined using a while-loop instead of a for-loop, then we must create a new example program if we wish to extract both the while- and for-loop variation.

If the surrounding world of a tutorial is often subject to change, then it is likely that the example programs of the tutorial are affected by these changes. In addition, some example programs can grow quite large and complex. For example, consider a framework tutorial or a tutorial like The Java Trail [10]. Every time the surrounding world is changed, we need to make sure that our example programs still compile and execute. Going through all example programs is a tedious task, especially if several variations exist. When the example programs have been brought into accord with the surrounding world, then the example programs and program tutorials must be synchronized.

Under the conditions described above, it becomes tedious to manually keep example programs and tutorial *consistent*. If changes happen often, some kind of change detection tool

would be appreciated. Hence, we need a tool that supports the programmer when creating tutorials as well as helps keeping the program tutorial consistent with both example program and the surrounding world. In addition, based on experiences grained from reading various program tutorials we hypothesize that most variations are simple variations; hence we need a tool that can help manage simple variations and incremental explanations.

In the Literate Programming approach presented in [4] the only way to handle either simple or complex variations is to state each variation in full. However, incremental explanations are well supported as chunks of an example program can be presented in any order chosen by the tutorial author, but it is not possible to have repetition of source code. Instead we propose using Elucidative Programming as a vessel for a tool that supports programmers writing consistent program tutorials including simple variations. In Elucidative Programming documentation and source code exists as separate entities. In fact one of the main goals of Elucidative Programming is to keep documentation and program source separated [2]. This means that the same program fragments can be addressed any number of times from anywhere in the documentation. The tutorial support for Elucidative Programming proposed in this paper is therefore based on source code extracts. The example program fragments that are in-lined in a tutorial are specified using XML as explained in section 3. These *extract specifications* are placed in the tutorial source and when the tutorial is presented the actual source code fragments are extracted and placed in the designated places in the resulting online presentation.

3 The Source Code Extraction Language

Extract specifications are written using *the source code extraction language*. The language must therefore provide means for addressing syntactical elements of a program. However, arbitrary syntactically elements are not equally easy to address. In a language such as Java, we can address program abstractions such as methods, classes, fields, and variables. The scheme for this is can be seen in figure 3.

Using the scheme in figure 3, addressing the method `totalAmount` in the class `Entry` residing in package `budget` is expressed as follows: `budget.Entry@totalAmount()`. If instead the expression `budget.Entry@totalAmount()@result` is used we address a local variable within the method `totalAmount`.

```
<javaentityname> ::= <package> | <class> | <field> | <method> | <parameter> | <variable> | <mark>
<package> ::= <dotname>
<class> ::= [<package> "."] <dotname>
<field> ::= <class> "@" <name>
<method> ::= <class> "@" <methodname>
<parameter> ::= <method> "@" <name>
<variable> ::= <method> "@" <name>
<mark> ::= (<class> "@" < markname>) | (<method> "@" <markname>)
<methodname> ::= <name> "(" [ <params> ] ")"
<params> ::= <class> { "," <class> }
<dotname> ::= <name> { "." <name> }
<markname> ::= "<e:" <name> "/>"
<name> ::= letter { letter | digit }
```

Fig. 3. BNF for addressing of Java entities. Curly brackets, ’{’ and ’}’, means 0 or more elements. ’|’ means or. Hard brackets, ’[’ and ’]’ means the element is optional [20].

In program tutorials we often have an additional need for addressing individual lines and blocks, but these cannot be addressed directly as when addressing named program abstractions. In order to address these we provide two options:

- Simple counting - a number that indicate the number of blocks or lines within a given context.
- Source marker - the block or line is marked with a unique identifier - a label – in form of a special Java comment (called mark in figure 3).

One of the requirements of Elucidative Programming is to leave the source code unaffected [2], but adding source markers requires actual change in the source code. However, as source markers are comments that do not directly affect the source code we allow this. See figure 4 for an example.

```
public double totalAmount()
 {
        double result=0; //<e:varDecl_totalAmount/>
        for (int month=firstMonth; month<=lastMonth; month++)
           {
                  if (this.payThisMonth(month))   //<e:firstSelection/>
                     result +=_amount;
           }
        return result; //<e:return_totalAmount/>
 }
```

Fig. 4. An illustration of use of source markers. The method `totalAmount()` contains three source markers: `varDecl_totalAmount`, `firstSelection` and `return_totalAmount`. The source marker `firstSelection` can be used to address the if-block at that line, whereas the two other source markers just address single lines.

Usage of simple counting makes the tutorial very sensitive to changes in the example programs. If a program tutorial is referring to a specific line in the source code it will become inconsistent as soon as new lines are inserted above the line referred to. There is no easy way to detect such change automatically. We therefore recommend using source markers. Using the scheme in figure 3 supports keeping program tutorials consistent with example programs because the tutorial is only addressing named abstraction in the program. In addition we can detect when a program tutorial is addressing program abstractions or source markers that do not exist. For example, when a method changes signature any tutorial that was addressing the method will produce a warning as they are now addressing a non-existing method.

Furthermore, a programmer working on source code knows that changes to a code area marked by a source marker may have an effect in the documentation. The programmer can bring the elucidative program up in a web-browser and navigate to the source marker (for example using the index of program abstractions, called “Source Index” in figure 1). Clicking on the source marker produces a click-able list of all the places in the documentation addressing the source marker. These places in the documentation can then be explored - a program tutorial may for example be referring to the source marker and appropriate actions can be performed. Furthermore, the Java Elucidator will report a warning if a part of a tutorial is referring to a source marker that no longer exists.

An alternative approach is to transform Java source files into XML documents using a source markup language such as JavaML [26]. XPointer (and XLink) [27] can then be used to address specific parts of the marked up source code. This will result in equivalent types of extracts as in our approach. There now exists an implementation of XPointer (and XLink) that can be used in the Java Elucidator that we will consider using in the future (see section 4.1).

3.1 The Basic Structure of the Source Code Extraction Language

The source code extraction language is intended to be both intuitively easy to use and understand when addressing source code fragments. We have therefore chosen a simple scheme for the structure of *extract specifications* that follows the structure of a program (specifically Java programs). As we shall see, the scheme can be seen as simple *pattern-matching scheme*, and is useful when addressing and extracting different fragments of a class or a method.

An extract specification, is represented by an *inline element*, and describes a single source code extract. An inline element either addresses a class or a method. An inline element can contain a collection of *sub-elements* addressing individual parts of the class or method addressed by the inline element. The allowed sub-elements are **lines**, **blocks**, **fields**, and **methods**. In turn, the sub-elements block and method can also contain sub-elements: a method element can contain block and line elements, and a block element can contain line elements.

The source code extraction language is written in XML syntax [15], as this is the current document format used in the Java Elucidator. For example, the pattern describing the first line of the if-block marked `firstSelection` and the line marked `return_totalAmount` in the method `totalAmount` in the class `Entry`, in figure 4, is expressed as follows:

```
<inline href=”Entry”>
    <method href=”Entry@totalAmount()”>
        <block type=”if” number=”firstSelection”>
            <line from=”1” to=”1”></line>
        </block>
        <line from=”return_totalAmount” to=”return_totalAmount”></line>
    </method>
</inline>
```

The attributes `href` addresses a specific program abstraction in the source code following the scheme in figure 3. The attribute `number` in the block element can either be a source marker or an integer. In the example the appropriate integer value would be “1”, as we wish to address the first if-block appearing in the method. The line element contains some redundancy because the tool currently requires that both attributes `from` and `to` be specified. The method `totalAmount` can also be addressed directly in the outermost inline element by changing the attribute `href` to ”`Entry@totalAmount()`”. Hence, we can omit the method element.

As the example illustrates, the structure of the extract specification follows the structure of the program. The extract specification describes a pattern consisting of syntactical elements that can be found in the parse-tree of the addressed example program. The tutorial tool searches the parse-tree in order to find these elements. The elements are extracted from the source code utilizing information about token positions stored in the parse-tree. This approach provides precise information on the positions needed in order to extract program fragments, and it was easy to implement. The approach has also proven to be efficient enough for our needs in practice. For example, a typical tutorial document consisting of 20 extracts from two different source files (counting 200 lines of Java code) can be produced in less than 3 seconds (on average the Java Elucidator takes 0.5-1.0 seconds to produce a typical document). The tool support concerning source code extracts will be discussed in more detail in section 4.1.

3.2 Handling Variations

As defined, simple variations are produced by presenting fragments of the example program in different order and detail. We therefore provide a *level of detail* of inline- and sub-elements. Level of detail applies to inline, method, and block elements, and available levels of detail are:

header: only displays the signature of a class, method or block.
none: nothing is displayed, applies to classes, methods and blocks
body: omits signature and displays the body of a class, method or block.
full: the default and displays an entire class, method or block.

For example, extract 2,3 and 5 in figure 2 are specified by:

```
<inline detail="header" href="Entry@totalAmount()">
    <line from="varDecl_totalAmount" to="varDecl_totalAmount"></line>
    <line from="return_totalAmount" to="return_totalAmount"></line>
</inline>

<inline detail="none" href="Entry@totalAmount()">
    <block type="if" detail="full" number="firstSelection"></block>
</inline>

<inline detail="full" href="Entry@totalAmount()"></inline>
```

The three inline elements address the same method (`Entry@totalAmount()`), but with different levels of detail. As can be seen in figure 2, this results in three simple variants of the method with repetition of source code. The first displays the signature the method and a number of lines (indicated by source markers). The second displays an if-block and nothing else of the method. The third displays the method in full detail. Using these levels of detail one can for example also produce class overviews showing only selected methods and fields in selected detail.

By combining the addressing scheme presented in section 3 and 3.1 with levels of detail we provide a tool that let the tutorial author choose which parts of an example program to extract and the order in which they are to be presented. This enables the tutorial writer to produce tutorials containing simple variations and repetitions, for example useful when explaining an example program in an incremental manner. These simple variations are small views of an example program, and although they are not full programs themselves a number of simple variations may constitute a variation of an entire example program. If there is an explicit need for the reader to be able to compile and execute such a variation then it must be stated in full as an individual example program.

In the BetaSIM tutorial [4] a full example program is provided for each variation. However, this can result in unnecessary and unwanted redundancy, as even a simple variation will require an entire example program to be stated in full. However, Østerbye suggests building a specialized *noweb* tool that can handle some kind of versioning scheme. Such a scheme will allow the tutorial writer to define different chunks with the same name but with different version numbers. In this way chunks can be extended or redefined. Running a specialized “tangling” program on the tutorial source can then produce different variations of an example program. Such a scheme will allow both simple and complex variations without redundancy, but with references to chunks instead of repetitions of source code.

Our current tutorial tool has no support for complex variations. For example, if a part of an example program is incrementally redefined, then we have no way of extracting the different variations from a given example program. Hence, each complex variation will result in a fully stated example program.

The source code extraction language, seen in figure 5, is still evolving in order to attend to different needs as we discover them. For example, local variable declarations are handled by

the line element, but could very well have a specialized “variable declaration“ element. The current language – as described in this paper - has been used to construct three tutorials, and has so far been found adequate. This is discussed in more detail in section 5.

Following the pattern scheme described in section 3.1 and the addressing scheme in section 3, the following summarizes the source code extraction language:

Class: <inline href=”A” detail=”D”></inline>
Extracts the class addressed by A and present it with detail D, where D is either: full, header, body or none.

Method: <method href=”A” detail=”D”></method>
Extracts the method addressed by A and present it with detail D, where D is either: full, header, body or none. Alternatively the inline element can be used.

Block: <block number=”N” detail=”D” type=”T” part=”P”></block>
Extracts the block addressed by N, where N is either a number or a source marker. The available block types are `if`, `try`, `for` and `while`. Hence, if N is `1` and type is `if`, the block element will extract the first encountered if-block within the root inline element. The part P indicates which part to be extracted of the block. For example, for a try-block one can either extract everything, only the try part or a specified catch-part. Detail D is either: full, header, body or none. To illustrate this consider the following example:

```
<inline detail=”none” href=”aClass@aMethod”>
        <block type=”try” detail=”body” part=”IOException” number=”firstTry”>
        </block>
</inline>
```

The above inline extract specification will cause an extraction of the catch part dealing with the exception called `IOException` in the try-block label with the source marker firstTry.

Line: <line from=”F” to=”T”></line>
Extracts the line addressed by F and T, where F and T are either numbers or source markers.

Fig. 5. The complete source code extraction language.

4 An Elucidative Tool For Creating Program Tutorials

The current tutorial tool includes a presentation generator that produces the online tutorials utilizing a source code extractor. The source code extractor is described in section 4.1, and the presentation generator is described in section 4.2.

The current Java Elucidator utilizes a database for storing information on program abstractions as well as abstractions in the documentation (links, sections and documents). Information in the database is synchronized with the actual program source files and documentation files by running an *abstraction*. After each abstraction it is easy to check whether any links in the documentation files are addressing non-existing program abstractions.

At editor level the Java Elucidator is supported by Emacs [13][14]. The editor support provides convenient look-up features that assist the programmer when addressing source code

from the tutorial source. These look-up features use information from the last abstraction. In addition, after each abstraction, a list of “dead link” links is provided. This list includes all inline- and sub-elements addressing non-existing program abstractions or source markers.

4.1 Extracting Source Code

As described in section 3.1 the current implementation of the source code extractor is based on a simple pattern-matching approach. The input to the extractor is inline elements, as the ones found in section 3.1 and 3.2. A modified version of the Kopi Java Compiler[1] [25] is used to build an abstract parse-tree of the addressed example program containing position information on all relevant Java tokens. The extractor tries to find a part of the parse-tree that matches the root of the input (a class or a method), and following search for sub-elements of the inline element in the parse-tree fragment. During this descend in the parse-tree, the extractor records all start and end positions of relevant elements in the source code. This approach makes the extractor precise enough to finally extract the needed parts of an example program. If there are parts of the input that does not match any part of the parse tree during extraction an error message is displayed (see section 4.2).

Although it is a different domain, source code extracts are often used as a reverse-engineering tool. One example is the lexical source model extraction (LSME) approach presented in [16]. The approach is based on regular expressions. LSME enables programmers examining a program to write a lexical specification describing the information to extract as a source model. A source model is for example: include-file dependences, call graphs, cross-reference lists, and program dependence graphs. Using this specification, the LSME system produces a source model from the program in question. The approach is flexible and tolerant, meaning that many types of input can be used and that input source code need not compile. In addition, this approach is likely to be somewhat faster than our approach.

However, the approach is not precise enough for our needs, as it is approximate – not all intended constructs may be extracted, and some unintended constructs may be extracted [16]. The LSME is not well suited for extracting fine-grained statement oriented source models. For example, we need to extract entire constructs, such as methods and blocks, which is difficult to express using the kind of regular expressions in [16] because we need to match balancing elements (curly brackets in Java). In addition, in order to extract single lines from an example program one must express this in a regular expression using a descriptive part of the specific line as a string.

As mentioned in section 3 an alternative approach would be to use XML marked up source code, XLink to specify (bi-directional) source code links, XPointer to specify the source code fragments to be extracted, and XSL for the presentation. Currently we are using XML and XSL but in the future we will examine possibilities of using XLink in order to reduce or eliminate the need for a database - depending on whether the various look-up features and “dead link” detection can still be provided and on whether performance is degraded. In addition, we can allow the user to use XPointer to express which source code fragments to extract. However, as our current extraction language is simple and intuitive it might be more attractive than XPointer. XPointer expressions are a bit more complicate to write as they deal with XML documents in general and not Java source. In this case the source code extraction language can simply be transformed into equivalent XPointer expressions.

[1] The Kopi Java Compiler has been modified in order to store more information on token positions than the original parser. In addition, our version of the parser ignores a range of compiler errors, such as missing imports.

4.2 Presenting Elucidative Program Tutorials

Elucidative program tutorials are presented in an ordinary web-browser. Figure 6 gives an example of an online elucidative program tutorial. The typical Elucidative Programming setup is two-framed where the left side displays documentation and the right side displays the source code. In the case of program tutorials the left side displays the actual tutorial containing tutorial text with interleaved source code extracts (dark boxes). If any error occurs during extraction this is reported. This happens if an extract specification is referring to a non-existing element, for example if referring to a for-loop that no longer exists or has been changed to a while-loop. Such errors cannot be detected during abstraction, but the tutorial writer can consult the program tutorial in a web-browser in order to ensure that no errors are reported. As an addition to the tutorial tool we could supply an error-checker that can be run from the editor.

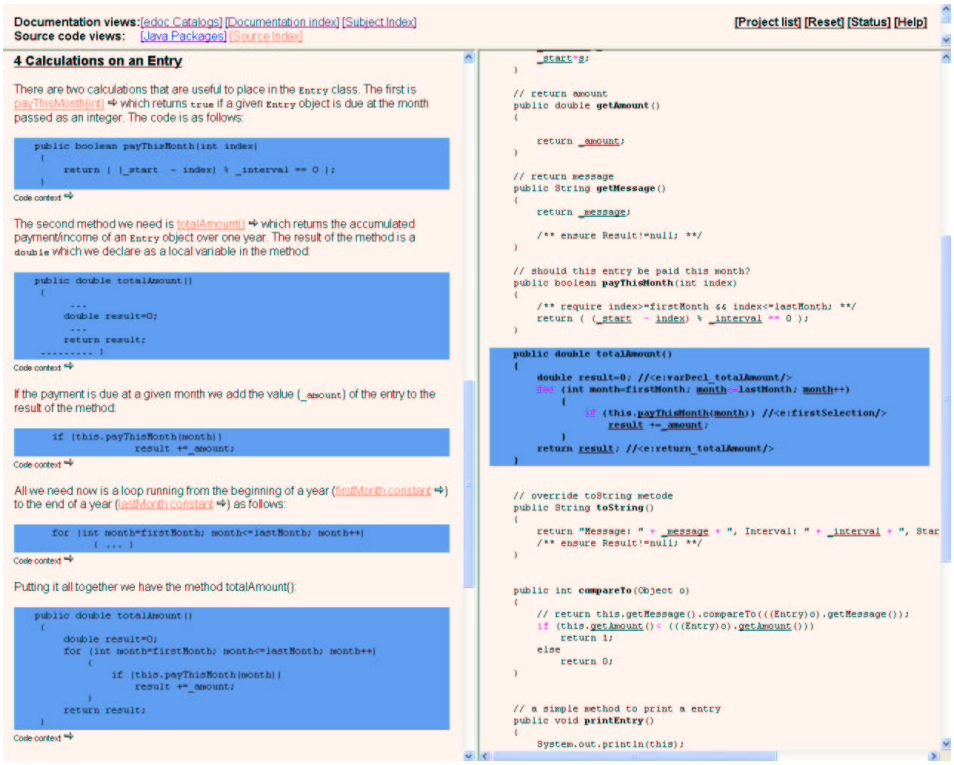


Fig. 6. An example of an online presentation of an elucidative program tutorial.

As seen in figure 6, two types of references are used from program tutorial to example program: extract specifications rendered as *in-lined* source code extracts and *ordinary* source code links (seen in the tutorials text together with a right-arrow).

Ordinary source code links from the tutorial text (the documentation) to source code are rendered as bi-directional links anchored at both source and destination. Using these links the reader can leave the tutorial in order to explore a program abstraction mentioned in the tutorial. We often use the term navigational proximity for this close relation between documentation and program [8]. The two-framed setup helps the reader to keep the context of the explanation while navigating to source code. However, should the reader choose to follow any link from a program abstraction to other parts of the documentation (e.g. other tutorials) the reader will entirely leave the context of the tutorial. In order to return the reader must use the back-button in the web-browser.

Compared to ordinary source code links the in-lined source code extracts relieves the reader from navigating from the tutorial text to the relevant place in the source code. A trip that often will disturb the reading. If the reader must constantly switch from a tutorial to an example program in order to follow references then the reader will be constantly interrupted and risk losing concentration. Such switching between "views" should be avoided as argued in [23], which addresses a hierarchy of cognitive issues to consider during design of software exploration tools.

Furthermore, a source code extract is always accompanied by a link to the source code extract in its original context. This is illustrated in figure 6, where a link called "code context" is seen beneath each source code extract. The result of selecting the last "code context" is seen in the right side of figure 6, where the relevant source code is found in a box similar to the source code box found in the tutorial text. This makes it easy to find the source code extract in the original context. For example, a source extract just addressing a single line can be very difficult to find in the body on an example program.

The presentation generator uses the same parser as the source code extractor in order to generate presentations of entire example program files. The tool utilizes knowledge about the position of program abstractions in order to produce hypertext presentations of example programs. In these presentations[2] links are provided from applied names to their definition, from program abstractions to documentation addressing a specific program abstraction (e.g. tutorials). If JavaDoc [22] is used to produce interface documentation of an example program links are provided from program abstractions to their respective documents in the interface documentation. In addition, links are provided from selected keywords to their definition in The Java Language Specification [21]. Links to The Java Language Specification are optional and were introduced because the audience of our first tutorials is students that have just started learning basic object oriented programming in Java, but this feature can in general be useful for all kinds of tutorials readers.

5 Discussion

We have not yet made any reel life experiments in order to evaluate the usefulness of the tutorial tool. However, the author of this paper has used the tool to create three program tutorials. Two of these explain proposed solutions to two student assignments. Each tutorial explains 5-6 classes consisting of 500-700 lines of Java code. The goal of producing the two tutorials was to test whether it is possible to construct meaningful tutorials using our Elucidative approach. The two example tutorials are available on the web via [11][3].

We see the tool as a promising way to support consistency between a program tutorial and its example programs. Fragments of example programs are extracted by addressing named abstractions, such as classes and methods. As long as these are not deleted or renamed then the "right" fragments will still be extracted after an example program has been changed. This is also the case when addressing individual lines and code blocks as long as source markers are used consistently - providing that the source markers themselves are not deleted, renamed or moved.

In addition, the "dead link" detection at editor level is a very useful aid as it pinpoints extract specification that are addressing non-existing abstractions. Also, during the evolution of the Java Elucidator "dead link" detection has proven very useful for checking the consistency between the internal documentation and the source code of the Java Elucidator. In addition, we have performed two experiments with elucidative programming where similar experiences were gained [28][29].

---

[2] The Java Elucidator can produce two types of presentations: dynamic and static. All link features described are available in dynamic mode, but some of them are not present in static mode.

[3] Currently best viewed with Internet Explorer 6.0 and Mozilla.

Furthermore, we find the source code extraction language intuitively easy to understand and use. The first elucidative program tutorial we made was a replica of a part of the Java Trail [10]. The goal of this was to test whether we can extract at least the same types of source code fragments as the ones found in the Java Trail. This was possible and the conversion was quickly done, as it was easy to come up with the needed source extract specifications. In addition, the Java Trail mainly deals with simple variations.

So far our support for simple variations have provided the flexibility needed in order to create program tutorials that explain example programs in an incremental manner. Should we in the future discover a need for supporting complex variations we will need to extend our current tool in order to handle this.

The strong side of the tool is that it supports incremental production of program tutorials and program and tutorial can evolve independently. The tutorials themselves are an alternative to traditional tutorials as we present a tutorial side-by-side with presentations of source code including presentations of source code fragments in their original context.

However, we have only experimented with constructing small tutorials written by the author (and colleagues). In order to evaluate the tool we need to perform reel experiments involving other tutorial writers and readers. Such an experiment could involve two groups of tutorial authors, one group using our tool and the other using "copy & paste". The tutorial authors are handed an example program at the beginning of the experiment but when the authors are done with the tutorial they are handed a new version of the example programs (with some selected changes). The tutorials must be changed accordingly. In this way we can measure and compare time spent on creating a tutorial and time spent on changing a tutorial. In addition, we can evaluate the usability of the tutorial tool (for example through interviews or "think aloud" experiments).

It could also be interesting to run experiments in order to evaluate the usefulness of the tutorials themselves. It should be possible to find examples of experiments with tutorials in the literature, and use these as inspiration.

Although our experiments are limited we already see potential in using the tool to produce tutorials that can be used in educational contexts. As argued by Sametinger in [18] it is not just important to teach students how to write documentation, we also need to expose them to well documented systems. Well documented (high-quality) systems can and should be used as examples in education in order to transfer design knowledge to the students. Sametinger recommends using concepts like literate programming, hypertext and object oriented documentation, as they are better-suited vessels than conventional documentation. Our tutorial support can be seen as such a vessel. We have chosen a solution that keeps us inside the Elucidative Programming paradigm, which is closely related to literate programming. In addition we utilize hypertext in order to provide easy navigation between example programs and their explanation. The result is program tutorials that are easy to study and check for consistency and completeness [18]. By exposing our students to elucidative program tutorials we can transfer design knowledge as well as demonstrate the importance of documentation. A program tutorial can also lead the students from a topic of interest in the tutorial to more in-depth documentation of a library, framework, or program (e.g. internal documentation).

As a side note, we also provide the option of saving all source code extracts as individual files. We have experimented with using these files as input to the LENO tool (a WEB based lecture note tool [24]), when building lecture slides. This means that the source code fragments used in a lecture can be automatically extracted from an example program and placed in the teaching material. This provides consistency in the teaching material as a whole, but also between an example program and its various presentations.

Furthermore, as we are using an ordinary web-browser and because we allow inclusion of HTML elements in our tutorials we can include multimedia and interactivity. For example, consider teaching a topic like recursion. An elucidative program tutorial can explain exemplar programs illustrating use of recursion by mixing source code extracts, text and interactive elements illustrating recursion (e.g. Java Applets or Flash animations).

As argued in [9], tutorials may play an active role in internal documentation. For example, during development it can be valuable to use tutorials as a common understanding of how a library is intended to be used. Furthermore, during construction of a framework potential problems can be detected at an early stage if example programs are written and explained during development. It can be very useful to apply user scenarios in order to guide a design process [17]. This can also apply to user scenarios in form of example programs illustrating how to use a library or a framework. Integrating such scenarios in the internal documentation makes them even more valuable.

As explained in section 1, due to the "paradox of sense making" people like to interact rather than study. This means that programmers are likely to make a lot of errors while exploring a library or framework. In [4] Østerbye handles this by providing the programmer with a set of training wheels in form of elaborate error checking and error messages. When programmers are "trying things out" they will get elaborate error message explaining them what they did wrong and how to correct the problem. As our approach is thought as a more general tool, we have not yet considered such error message support. However, if we wish to use our program tutorial tool and Elucidative Programming to produce end-user documentation of framework then such support should be considered more carefully.

6 Conclusion

In this paper we have presented a tool that supports authoring of program tutorials. Based on extract specifications in the tutorial source, fragments of example programs are extracted and interwoven into the tutorial text.

In order to help ensure consistency between program tutorial and example programs we have provided a mechanism that allows us to address named program abstractions, such as classes and methods. The addressing mechanism also allows us to address markers placed in the source code, providing the option of addressing individual code lines and code blocks. Using the source code extraction language presented in this paper one can specify the fragments of a program that are to be extracted. When using program abstractions to address fragments, as opposed to addressing fragments using line numbers or search strings, the extract specifications will not need updating when the program is changed – providing that addressed program abstractions are not deleted or renamed.

The tool also helps keeping program tutorial and example program consistent through "dead-link" detection. If a part of an extract specification is referring to a non-existing program abstraction or does not match a part of the example program then this is reported.

In addition, the tool enables the tutorial writer to create program tutorials that can present simple variations of example programs and include repetition of source code. Repetition of source code allows having different tutorials that are addressing the same example programs. Furthermore, a program tutorial can use repetition of source code to explain matters in an incremental manner by stepwise adding more and more complexity to the previous extract.

The program tutorials are presented in an ordinary web-browser. Each source code extract is rendered with a link to a presentation of the extract in its original context in the full example program. In this presentation the selected source code extract is highlighted making it easy to find. Furthermore, the curious tutorial reader can follow links from program abstractions in the example program to relevant documents such as other tutorials, internal documentation, and interface documentation (produced using JavaDoc).

Although we already see good reasons for applying our tool in order to produce elucidative program tutorials, especially in educational contexts, we will in the future conduct more experiments. These experiments will involve both tutorial readers and writers in order to examine the usefulness of the program tutorial presentations, the source code extraction language, and the editor support.

References

1. Knuth, D. E.: Literate Programming. The Computer Journal, vol. 27:97-111, May 1984.
2. Nørmark, K.: Requirements for an Elucidative Programming Environment. International Workshop on Program Comprehension 2000, pp. 1-13.
3. Nørmark, K.: Elucidative programming. Nordic Journal of Computing, 7(2):87–105, 2000.
4. Østerbye, K.: Minimalist documentation of frameworks. Presented at 3rd ECCOP'99 Workshop on Experiences in Object-Oriented Reengineering, 1999.
5. Carrol, J. M.: The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill. The MIT Press, 1990.
6. Selber, S. A. & Johnson-Eiloal, J. & Mehlenbacher, B.: On-line support systems: Tutorials, documentation, and help. In "Handbook of Computer Science and Engineering", ed. Tucker, A. B., 1619-43, Boca Raton, FL: CRC Press.
7. Vestdam, T.: Documentation threads - presentation of fragmented documentation. Nordic Journal of Computing, 7(2):106–125, 2000.
8. Nørmark, K. & Andersen, M. R. & Christensen, C. N. & Kumar, V. & Staun-Pedersen, S. & Sørensen, K. L.: Elucidative programming in java. In Proceedings on the eighteenth annual international conference on Computer documentation (SIGDOC). ACM, September 2000.
9. Vestdam, T., Nørmark, K.: Aspects of Internal Program Documentation - an Elucidative Perspective. To be presented at International Workshop on Program Comprehension 2002.
10. The Java Tutorial (Java Trails), http://java.sun.com/docs/books/tutorial/
11. Examples of elucidative program tutorials, 2002, http://www.cs.auc.dk/~odin/api/example/
12. Ramsey, N.: Literate programming simplified. IEEE software, 11(5):97-105,1994.
13. Stallman, R.. Emacs: The extensible, customizable, selfdocumenting display editor. In D. Barstow, H. Shrobe, and E. Sandewall, editors, Interactive Programming Environments, pages 300–325. McGraw-Hill, 1984.
14. Stallman, R.: GNU Emacs manual. The Free Software Fundation Inc, June 1985.
15. Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation 6 October 2000, http://www.w3c.org/XML
16. Murphey, G. C. & Notkin, D.: Lightweight Lexical Source Model Extraction. ACM Transactions on Software Engineering and Methodology, Vol. 5, No. 3, July 1996, Pages 262–292.
17. Carroll, J.: Making use a design representation. Communications of the ACM, 37(12): pages 29–35, December 1994.
18. Sametinger, J.: The Role of Documentation in Programmer Training. Programming Languages: Experiences and Practice, Mark Woodman (Ed.), Chapman & Hall, 1994.
19. Butler, G. & Dénommée, P.: Documenting frameworks. Building Application Frameworks: Object-Oriented Foundations of Framework Design, pages 495–504, September 1999.
20. Andersen, M. R. & Christensen, C. N. & Kumar, V. & Staun-Pedersen, S. & Sørensen, K. L.: The elucidator – for Java. Preliminary masters thesis report, January 2000. Available from http://dopu.cs.auc.dk
21. Steele, G. & Gosling, J., & Bracha, G.: The Java Language Specification. The Java Series, Addison-Wesley, second edition, 2000.
22. Friendly, L.: The design of distributed hyperlinked programming documentation. In S. Frass, F. Garzotto, T. Isakowitz, J. Nanard, and M. Nanard, editors, Proceedings of the International Workshop on Hypermedia Design (IWHD'95), Montpellier, France, 1995.
23. Storey, M.-A.D. & Fracchia, F. & Muller, H.. Cognitive design elements to support the construction of a mental model during software visualization. In Proceedings of the Fifth International Workshop on Program Comprehension, pages 17–28, March 1997.
24. Nørmark, K. WEB Based Lecture Notes – The LENO Approach. Unpublished paper available via http://www.cs.auc.dk/~normark/laml/, November 2001
25. Gay-Para, V. & Divoky, W. & Graf, T. & Lemonnier, A. G. & Wais, E. Kopi Java Compiler. 1999, http://www.dms.at/kopi/kjc.html
26. Badros, G. J.: JavaML: A Markup Language for Java Source Code. Ninth International World Wide Web Conference, Amsterdam, May 2000.
27. W3C XML Pointer, XML Base and XML Linking - XML Linking Language (XLink) Recommendation 1.0, XML Pointer Language (XPointer) Candidate Recommendation, http://www.w3c.org/XML/Linking, 2002.
28. Vestdam, T.: Introducing elucidative programming in student projects. Unpublished paper available via http://dopu.cs.auc.dk, January 2001.
29. Andersen, M. R. & Christensen, C. N.: Evaluating elucidative programming in an industrial setting. Unpublished paper available via http://dopu.cs.auc.dk, December 2000.