A GRAPH DESCRIPTION LANGUAGE FOR BLIND PROGRAMMERS

Dale E. Parson, Genevieve Smith, Andrew Wernicki
Department of Computer Science and Information Technology, Kutztown University of PA
parson@kutztown.edu

ABSTRACT

Sighted people often overestimate the usefulness of tactile diagrams for blind people who need to perceive and comprehend entities and relationships in visual structural graphs. Sighted software developers can acquire an overview of the structure of a Unified Modeling Language (UML) design diagram after a quick visual scan. Blind developers using tactile diagrams are impeded by the serial nature of scanning these diagrams with their fingers. However, blind programmers are experts at using text-to-speech screen readers to verbalize lines of existing code at a very rapid rate. Comprehending and debugging code with a screen reader happens extremely quickly because blind developers have a lot of practice using such tools. This paper reports on using a block-structured diagram description language with syntax and keywords similar to modern programming languages to capture and communicate entities and relationships in standard UML diagram types. Prototype software tools include utilities for verifying textual UML models, for converting between textual representations useful to blind versus sighted programmers, and for generating visual diagrams for communications with colleagues. Creating this language in support of a blind student in an object-oriented design course helped to accelerate acquisition and communication of design concepts among students and the instructor.

KEYWORDS

blind programmers, blind developers, graph description language, tactile diagrams, Unified Modeling Language

1. Introduction

1.1 An instructor's perspective

This report grows out of experiences supporting one of the authors, a blind student (Smith), in a graduate course in advanced object-oriented design and programming at Kutztown University in spring 2016. The course makes heavy use of numerous diagram types from the Unified Modeling Language (UML) [1] in reverse-engineering an existing code base for project 1 and in designing a system built in several subsequent projects. The instructor (Parson) learned of difficulties in obtaining tactile diagrams for a previous course in a timely manner during the preceding fall semester. By ordering a half semester ahead, we managed to acquire tactile diagrams for all of the illustrations for the UML guide used in the class [2] by the beginning of the term.

However, it became clear to the instructor at the very start of going over UML diagrams in class, that the serial cognitive bottleneck imposed by using tactile diagrams was very likely to cause two problems: It would slow down acquisition of diagrammed design information by the student, which in turn would slow down the pace of the class. Tactile diagrams require serial scanning, with no holistic picture of an entire design, and they can span multiple pages, even for visual diagrams that do not, thanks to annotations that expand the space requirements of tactile diagrams.

Thankfully, the instructor had also observed this student using the JAWS screen reader to capture and debug programs [3]. She can scan sequential lines of code text and focus in on compile errors with amazing speed. The instructor had previous experience supporting a textual description language for writing event-driven simulation models as UML state machine diagrams [4]. A critical observation is the fact that the large majority of UML diagramming constructs are in fact text entries for entity names, methods, data, and various tags. Fully graphical entities are of only a few kinds, primarily nesting boxes and arrowed links. A textual UML graph description language needs only to invent constructs for the relatively few inherently graphical constructs in UML diagrams.

1.2 A blind programming student's perspective

Transforming visual concepts into nonvisual mediums used by the blind is often a cumbersome, inefficient, and difficult task, regardless of the alternative medium employed. The creation of several alternative systems, including tactile diagrams, auditory illustrations, and methods utilizing both auditory and tactile feedback pose their own limitations and challenges. First, new systems often impose a learning curve for both the creator and user, creating unfair burden for everyone involved. Second, many of these systems are either cost-prohibitive to students and universities, or require manual labor to implement. Third, though guidelines have been published,

no standard among diagram users or their creators exists. Fourth, few systems allow the blind user to create diagrams independently, hindering progress in a competitive visual society. Furthermore, comprehension of abstract concepts, especially when conveyed tactilely, coupled with an overall negative disposition toward diagrams, causes many users to feel frustrated and overwhelmed [5]. Many factors affect the choice of medium to display nonvisual material.

Likewise, the importance of determining the usability of any alternative diagram system cannot be overstated, especially when assessing perhaps the most common of the three mediums: tactile diagrams. Tactile diagrams incorporate enlarged raised images that the blind perceive with their fingertips. Though expensive to purchase, using imaging tools and a braille embosser, tactile diagrams are relatively straightforward to produce. Furthermore, their usability advantages make them a practical option in some circumstances. Relatively simple diagrams offer blind users a tangible method they can perceive, helping to clarify unfamiliar abstract concepts by making them more concrete. Tactile mediums reduce the burden of mental mapping, because blind users are able to refer back to the diagram repeatedly and with ease. Equally important, tactile methods help preserve and convey spacial information, though their limitations also contribute to some spacial inaccuracies. Tactile diagrams are most beneficial while modeling simple concepts that represent tangible objects.

Nevertheless, the drawbacks of tactile diagrams may cause users to seek alternative systems of communication. The limitations of tactile perception become apparent when modeling complicated diagrams and abstract concepts. The lack of variety that the tactile sense provides, in combination with the inability to perceive large pieces of the diagram at once, hinder information synthesis and cause strain on user memory. In addition, detailed diagrams must be spread across several pages, further taxing the user. In an effort to reduce clutter in these diagrams, venders make labeled information accessible via a numbered key. The reduction in clutter is offset by the inefficiency of turning pages to find the label's meaning. Moreover, the time taken to trace, synthesize, process, and develop a mental map of a complex tactile diagram is slow, cumbersome, and exhausting. In an effort to address shortcomings of other systems including tactile perception, we have created an auditory text-based system to represent UML diagrams.

2. Related work

Our initial thought before working with tactile diagrams was to experiment with the creation of tactile diagrams using raised, tactile printing or 3D printing [6]. The negative experiences with tactile diagrams related in the Introduction led us to investigate other mechanisms.

The TeDUB system appears to be the most ambitious and thoroughly supported software system for enabling blind software developers to use UML [7], being supported by grants from the European Union [8]. Its purpose is to allow navigation of visual UML diagrams by tagging them with additional meta-data that allows blind users to navigate diagram contents using special, text-oriented navigation tools. Its scope exceeds that of the present work by allowing users to extract information from existing tagged diagrams. However, from the perspective of the present work, TeDUB's provision of a specialized structure navigator that does not utilize computer text readers that are heavily used by blind users makes it usage appear cumbersome and unnecessarily novel.

A survey of means for communicating UML diagrams to blind developers [9] lists these commonly-used techniques: 1) *manual methods* such as a stencil embossing kit; 2) Braille embossers and stereocopying; 3) tactile display; 4) tactile diagram plus audio; and 5) verbal description only by another person. Unfortunately, this otherwise excellent survey makes the following assertion about using text-based diagram descriptions with a computer reader: "A variety of approaches can be used to describe the relationships between, and the contents of UML objects. Although this approach works well for blind programmers, it does not fit well into the practices, standards and abilities of sighted programmers. UML diagrams are supposed to be diagrams - not audio presentations or verbal tours through a software design. UML users expect to get information on many levels simultaneously - relationships, structure, details; even standards are expressed immediately to a sighted programmer when they see a UML diagram. Sighted programmers are not capable of the long-term memory and cognitive pattern building abilities that blind programmers are forced to have. A blind programmer using this technology can not expect advancement is a software company who insist on standard UML notation." This position denies any need for accessibility accommodation by a development organization. It ignores the value that good software developers who happen to be blind can bring to an organization, and its assertions about sighted programmers underdeveloped long-term memory and an organization's prerogative to discriminate against blind programmers are ignorant and dangerous.

It was the instructor's experience creating a textual description language for UML state machine diagrams used as simulation models [4], and the recognition that UML diagrams consist largely of text, that led to the creation of the notation presented in this report. In answering the reservations of using non-visual diagram descriptions cited in the previous paragraph, it is important to point out two facts. First, in the system presented here, it is possible to generate UML-compliant visual diagrams for communication with colleagues by generating visual diagrams from our structured textual descriptions using a toolset such as Graphviz [10].

Second, while going from graphs to structural descriptions is more problematic when the graphs are in a strictly visual form such as an image file, such a flat representation of UML diagrams falls far short of the state of the art of graphical design tools for software. The instructor was formerly a software architect and developer with Bell Laboratories whose projects in the late 1980s and early 1990s included work on a proprietary graphical schematic capture tool called SCHEMA that, while supporting visual interaction, stored all of its design data in textual descriptions. Given the stereotyped nature of UML diagrams, in which the boxes, other containers, and links take very stereotyped form, there is no technical reason that textual and visual UML representations and storage formats cannot be interchangeable. Creating tools that can support sighted and blind developers in using visual and textual diagrams is a straightforward matter of software technology.

The effort reported here relies strictly on textual capture of diagrams. It does have the ability to generate visual graphs. It represents progress towards a toolset that could support design capture in either modality, generating visual graphs from structured text descriptions, and generating text descriptions from annotated visual graphs. A complete toolset would essentially be a UML CASE tool (Computer Aided Software Engineering) for sighted and blind developers alike.

3. A Graph Description Language

3.1 Basic compiler technology

This compiler is somewhat more complex than the compiler for UML state machine diagrams [4] in that it compiles multiple diagram types. But, unlike the previous effort, the toolset does not include a run-time simulator. The compiler is written in Python, with a front-end scanner-parser written using the PLY library that is a Python equivalent of the YACC parser generator for C language [11]. Coding the scanner, parser, and semantic checker in Python allows the front end to save symbol table and parse tree information in a textual file format.

While Python and PLY work well for rapid prototyping, Python is not well suited to blind programmers because, unlike C, C++ and Java that use curly braces to denote nested structures such as blocks of code, Python uses mandatory indentation of code. Indentation does not work well with many text-to-speech readers because they simply count the spaces out loud. They do not give a good feel for level of nesting because they simply enumerate spaces. In order to accommodate a blind developer, we have the front end save a symbol table and parse tree as a textual Python data structure, which we then run through a Jython program that reads the text file and translates it to a nested Java list of lists and strings using java.util.List, which it then writes to a serialized Java data file. Jython is a Java implementation of Python that has access to the complete Java class library [12,13]. A serialized Java data file is a binary data file that does not require a developer to implement a writer or reader for the structured file [14].

The downstream, back end plugins of the system read serialized Java data files containing symbol tables and parse trees, using Java as a programming language. There are presently three back ends. One emits a copy of an original textual model formatted for use with a blind reader. Accommodations include eliminating leading spaces and tabs for indentation, which result in cumbersome enumeration of spaces by an automated reader, and attachments of header comments with the header portions of container constructs. A second, alternative back end emits a copy of the original textual model formatted for a sighted developer, employing appropriate indentation. A third back end emits instructions for GraphViz construction of a visual diagram.

3.2 Notation patterns and example diagrams

Our graph description language at present includes the following types of UML diagrams: deployment, class, object, state machine, sequence diagrams. Addition of activity diagrams is pending; we used pseudo-code in the place of activity diagrams for the course. These are all of the diagram types used in our course, and include nested classes, generic classes, interface and implementation inheritance, active classes and objects that run their own threads, and other standard UML constructs.

Before showing two specific examples, we list a few basic rules in language design that arose through planning and experimentation with using the language.

Rule 1: Use { curly braces } to denote any kind of container in a UML diagram, preceded by the name of the construct that the box represents, for example “class” or “object”. Curly braces can nest for nested containers. Blind programmers are used to using curly braces for representing nested structure in a block-structured source language such as C or Java. Given the fact that UML diagrams consist almost entirely of containers and links annotated with text, curly braces cover essentially half of the graphical functions.

Rule 2: Do not use special symbols such as “->” for arrowed links because text-to-speech readers read such constructs as “dash greater” at a rate that slows reading. Such symbols are oriented primarily towards sighted developers. Use keywords instead. Discovering this rule required some trial and error usage of the language.

Rule 3: Borrow keywords from Java where they provide a good fit to the intended semantics, and make up keywords as necessary.

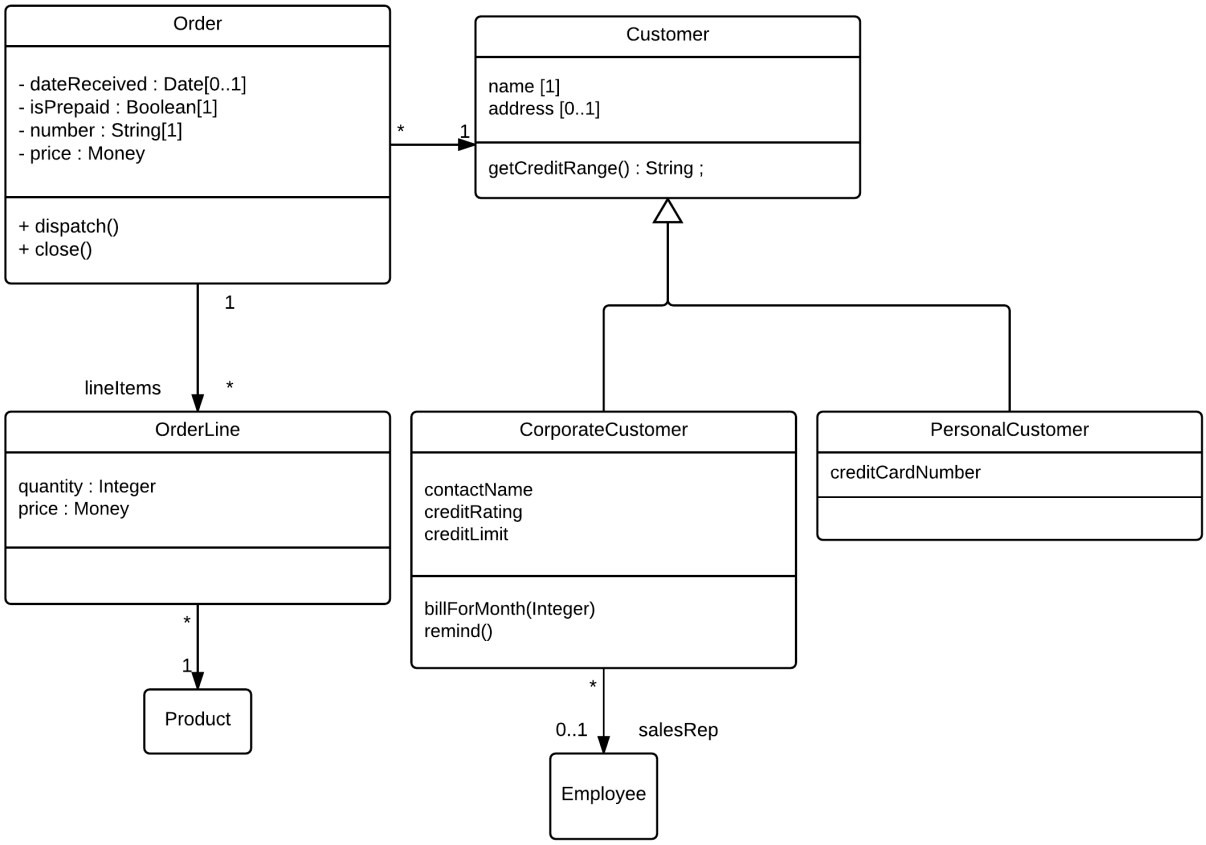

Figure 1: UML class diagram showing inheritance and association

Figure 1 shows a UML class diagram that is a modified version of Figure 3.1 in the UML textbook [2]. It includes several attributes (data fields), operations (methods), generalization (inheritance), visibility (+ for public and – for private within class Order), and directional association links with multiplicity and roles. Note that, other than containers and links, the information appears in the form of text. Other class diagram constructs including active and nested classes do not appear in this diagram, but our notation supports them.

What follows is our equivalent text notation, with comments removed for brevity.

```
classDiagram Figure1 {
   class Customer {
      name [1] ;
      address [0..1] ;
      getCreditRange() : String ;
   }
   class Employee {
   }
   class CorporateCustomer extends Customer {
      contactName ;
      creditRating ;
      creditLimit ;
      billForMonth(Integer);
      remind();
   } [*]uses[0..1] "salesrep" Employee ;
   class PersonalCustomer extends Customer {
      creditCardNumber ;
   }
   class Product {
   }
   class OrderLine {
      quantity : Integer ;
      price : Money ;
   } [*]uses[1] Product ;
   class Order {
      - dateReceived : Date[0..1];
      - isPrepaid : Boolean[1];
      - number : String[1];
      - price : Money;
      + dispatch();
      + close();
   } [*]uses[1]Customer, [1]uses[*] "lineItems"
OrderLine ;
}
```

Listing 1: Graph description for Figure 1

The two formats are so interchangeable that the instructor simply copied and pasted text between the graphical tool [15] and a text editor to prepare diagrams.

Using nesting {curly braces} to denote nesting containers is the only non-keyword counterpart to the set of several box-shaped UML graphical counterparts. We use keywords such as “active”, “static”, “abstract”, “class”, “interface”, and “object” (in object diagrams), and others to qualify the {}-delimited constructs, borrowing from Java where there is an appropriate Java counterpart. UML visibility, multiplicity, role, and similar qualifiers are text-based and fit easily into the language syntax.

Similarly, we use Java keywords “implements” and “extends” for inheritance (UML generalization). These appear as arrowed links with unfilled arrowheads in UML. Arrowed and non-arrowed links presented more of a problem. We initially tried using visual symbols “->”, “<-”, “<->”, and “--” as textual counterparts to UML directed and undirected associations, but readers such as JAWS read these as “dash greater”, for example, introducing auditory clutter. We settled on new keywords “uses”, “usedby”, “useboth”, and “usehuh” for the above four association types. There was much debate and experimentation before settling on these words. Note the use of “uses” with multiplicity and role tags on the appropriate sides of this keyword in Listing 1, and the correspondence to the visual diagram in Figure 1.

Listing 2 shows the textual Python parse tree that the compiler front end passes to Jython and Java back ends. It produces both indented and non-indented versions for sighted and blind programmers working on back end code, and it preserves everything in the source file, including comments, which do not appear here for brevity.

```
parsetree = \
('diagram:', ('classDiagram:', 'Figure1', '{',
   ('class-sequence:',
      ('class:', 'class', 'Customer',
         '{',
         ('class-contents:',
            ('attribute:', 'name', '[1]', ';'),
            ('attribute:', 'address', '[0..1]', ';'),
        ('method:', 'getCreditRange', '(', ')', ('type:', 'String'),
              ';')), '}'),
      ('class:', 'class', 'Employee', '{', '}'),
      ('class:', 'class', 'CorporateCustomer',
          ('extends', 'Customer'),
         '{' ('class-contents:',
            ('attribute:', 'contactName', ';'),
            ('attribute:', 'creditRating', ';'),
            ('attribute:', 'creditLimit', ';'),
            ('method:', 'billForMonth',
               '(',
               ('param-list:', ('Integer',)),
               ')', ';'),
               ('method:', 'remind', '(', ')', ';')),
            '}',
            ('association-list:',
              ('uses:', '[*]', '[0..1]', '"salesrep"', 'Employee'), ';'),
              ),
      ('class:', 'class', 'PersonalCustomer',
         ('extends', 'Customer'),
         '{', ('class-contents:',
            ('attribute:', 'creditCardNumber', ';')),
         '}'),
      ('class:', 'class', 'Product', '{', '}'),
      ('class:', 'class', 'OrderLine',
         '{', ('class-contents:',
            ('attribute:', 'quantity', ('type:', 'Integer'), ';'),
            ('attribute:', 'price', ('type:', 'Money'), ';')),
         '}',
         ('association-list:',
            ('uses:', '[*]', '[1]', 'Product'),
            ';')),
      ('class:', 'class', 'Order',
         '{', ('class-contents:',
            ('attribute:', 'dateReceived', ('type:', 'Date'),
               '[0..1]', ';'),
            ('attribute:', 'isPrepaid', ('type:', 'Boolean'),
               '[1]', ';'),
            ('attribute:', 'number', ('type:', 'String'),
               '[1]', ';'),
            ('attribute:', 'price', ('type:', 'Money'), ';'),
            ('method:', 'dispatch', '(', ')', ';'),
            ('method:', 'close', '(', ')', ';')),
         '}',
         ('association-list:',
            ('uses:', '[*]', '[1]', 'Customer'),
            ('uses:', '[1]', '[*]', '"lineItems"', 'OrderLine'),
            ';'),
            )),
  '}'))
```

Listing 2: Textual parse tree for Figure 1

Figure 2 shows the flat graphical representation of the class diagram that the back end of our toolset generates from the graph description of Listing 2. This back end uses a text-driven graph generator tool called PlantUML [16] that builds, in part, on GraphViz.

For completeness we illustrate a UML sequence diagram that shows method calls and return values among objects. Unlike a deployment, class, and object diagrams, which model structure, sequence and activity diagrams are example diagram types that model behavior. Listing 3 is the non-indented version that is more useful to a blind developer. As noted, our toolset’s back end generates both indented and non-indented versions of its diagram description files.

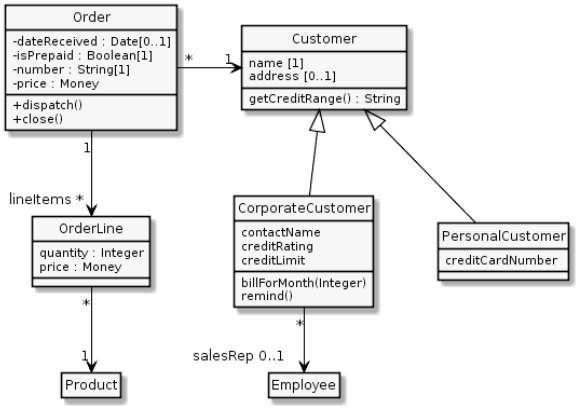

Figure 2: Class diagram generated from Listing 2

```
sequenceDiagram Figure3 {
object anOrder : Order {}
object anOrderLine : OrderLine {}
object aProduct : Product {}
object aCustomer : Customer {}
? calls anOrder.calculatePrice() ;
// "?" means call comes from outside displayed objects.
anOrder.calculatePrice() calls anOrderLine.getQuantity()
;
anOrder.calculatePrice() calls anOrderLine.getProduct() ;
anOrderLine.getProduct() returns aProduct to
anOrder.calculatePrice() ;
// aProduct is the return value to object anOrder from its
// method call to anOrderLine.getProduct().
anOrder.calculatePrice() calls
aProduct.getPricingDetails() ;
anOrder.calculatePrice() calls
anOrder.calculateBasePrice() ;
anOrder.calculatePrice() calls
anOrder.calculateDiscounts();
anOrder.calculateDiscounts() calls
aCustomer.getDiscountInfo();
}
```

Listing 3: Graph description for Figure 3

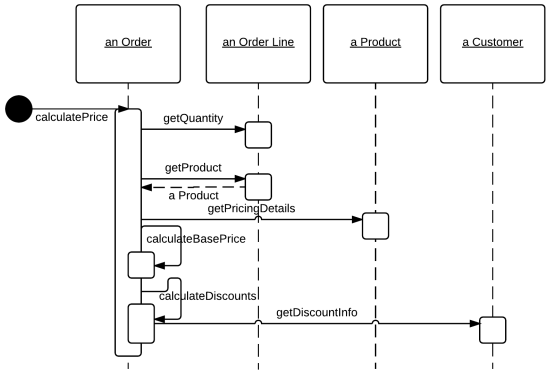

Figure 3: UML sequence showing calls and returns

3.3 Usability from an instructor's perspective

The instructor's primary criteria for success of this project are the satisfaction of the customer student and the ability to speed up the pace of the course. The next subsection addresses the former criterion. It is absolutely essential to have a blind developer involved in the development of a system such as this, because of the need for quick feedback when making decisions about language syntax, keywords, and useful tools.

This graph description language has done more than speed up the pace for one student. It provides an alternative means to discuss and review the constructs of UML diagrams, and it is easy and fast to capture. It requires only a text editor to capture a design. After getting practice using visual UML diagrams on assignments, several students switched to our graph description language because of its ease of use and speed.

3.4 Usability from a blind developer's perspective

The text description language uses familiar code-like structure, keywords, and syntax to represent each UML diagram. Language words like "class" and "extends," for instance, describe classes and their relationships to one another, function definitions identify functions – though parameters are structured differently than in Java – and braces identify the beginnings and closings of classes and the entire page. Minor differences occur where necessary in order to save space or preserve meaning. Public, private, and default specifiers are symbolized with plus, dash, and tilde respectively, parameter names are proceeded by a colon and then the parameter type, and function return types are written last in function specifications. The instructor and blind student both use standard computers to read and write the text description language, but the blind student uses a software program called a screen reader to access the text. Screen readers parse and interpret data and then return the information as synthesized text back to the user.

The primary benefit of our text description language is the reliance on already-familiar syntax and structure. Accessible software and tools often force blind users to learn a variety of commands, and, initially, the usability of the system is partially compromised by system memorization. Prior coding background prevented a frustrating transition, eliminating a learning curve of an entirely unfamiliar system. This, in turn, contributed to a smooth process when new syntax was introduced.

Another major benefit is the ability for blind users to produce diagrams independently. Unlike the majority of systems that required sighted intervention at some point in the process, the entire procedure – from diagram transcription to navigation – could be completed without any sighted assistance whatsoever. Independence is

paramount in a competitive visual society, and no diagram system should be considered complete without it. Furthermore, to aid sighted users, a project is under development that transforms a text diagram into an illustration. It is uncertain whether or not software can reliably translate a UML diagram back into text description language.

Lastly, except for the expense of a screen reader, our system is cost-effective, and minimal labor to produce text diagrams is involved. The exclusion of specialized tools and software eliminates the price factor entirely, and blind users now have access to open-source screen readers as well. I speculate that typing text is much faster than drawing diagrams, and any existing functions can be copy/pasted into them.

Though our system addresses the primary drawbacks of other comparable tools, it is not infallible. Blind users will still need to mental map large and detailed diagrams, though the efficient traversal will help offset this burden. . I hypothesize that mental mapping will always be necessary, regardless of the system, since neither auditory or tactile perception is comparable to sight. The benefits addressed above far outweigh this drawback.

Though most issues with the description language have been resolved, the process of refinement involved trial-and-error, and in some cases, the language is still a work-in-progress. The main challenge we faced required eliminating auditory clutter while preserving scannability. Screen readers navigate in a linear fashion, and users generally begin at the top of documents and work their way down. How much and the type of information spoken is controlled in a screen reader's verbosity and punctuation settings and is dictated by user preferences. Because coding syntax is essential to the language, my settings are set to speak most punctuation. To maintain efficiency and improve navigation speed, it was crucial to shorten any unnecessary clutter. This included reducing comments to a bear minimum and placing them at the ends of lines when absolutely necessary, allowing them to be skipped over more easily. Braces appear by themselves on separate lines. Additionally, indentation was eliminated because it served no practical purpose and because it only hindered the editing process.

The text description language has been, and will continue to be, an extremely useful tool to represent visual UML diagrams. Text descriptions have resolved primary issues of cost, system learning curves, and user independence. The system has also contributed positively to efficiency and speed. Though issues with mental mapping and transcription remain, the language has provided a platform that helps facilitate blind user inclusion.

2. Conclusions and future work

Our UML graph description language has proven itself to be a practical, useful tool set for both blind developers and their collaborators in software design and reverse engineering projects. It integrated seamlessly into the course on object-oriented software development, and can integrate just as seamless into professional software development.

The round trip between textual descriptions and visual diagrams is the big remaining hurdle. There is a prototype back end to the tools that generates flat visual diagram image files. The ideal goal is to have a visual structure editor for capturing diagrams using data structures and file formats that support round-trip editing in both the visual and textual domains. We plan to present our work to several commercial UML tool vendors, for example [15], after publication. We hope to find means to continue and extend this work into a round-trip, textual-visual toolset.

Acknowledgements

The authors would also like to express our appreciation for a student work time grant from the Kutztown University Research Committee.

References

[1] Object Management Group, UML 2.0, http://www.omg.org/spec/UML/2.0/, July 2005.

[2] M. Fowler, UML Distilled, Third Edition, Addison-Wesley, 2004.

[3] Freedom Scientific, JAWS Headquarters, http://www.freedomscientific.com/JAWSHQ/JAWSHeadquarters01.

[4] D. Parson, "A State Machine Language for the Undergraduate Operating Systems Course," *Proceedings of the 29th Annual Spring Conference of the Pennsylvania Computer and Information Science Educators* (PACISE) California University of PA, California, PA, April 4-5, 2014.

[5] Aldrich, F. K., & Sheppard, L. (2001). "Tactile graphics in school education: Perspectives from pupils." *British Journal of Visual Impairment*, 19(2), 69-73.

[6] C. Loitsch and G. Weber, "Viable Haptic UML for Blind People," *Proceedings of the 13th International Conference on Computers Helping People with Special Needs* - Volume Part II, July, 2012.

[7] King, Blenkhorn, Crombie, Dijkstra, Evans, and Wood, "Presenting UML Software Engineering Diagrams

to Blind People," *Proceedings of Computers Helping People with Special Needs*, 9th International Conference, ICCHP 2004, Paris, France, July 7-9, 2004.

[8] Horstmann, Hagen, King, and Schlieder, Automated interpretation and accessible presentation of technical diagrams for blind people, New Review of Hypermedia and Multimedia, Vol. 10, No. 2, December 2004.

[9] Baillie, Burmeister, and Hamlyn-Harris, "Web-based teaching: communicating technical diagrams with the vision impaired", *Proceedings of Multi-modal Content: Flexible, Re-useable and Accessible, the 2003 Australian Web Adaptability Initiative Conference* (OZeWAI), Bundoora, Victoria, Australia, December 2003.

[10] *Graphviz - Graph Visualization Software*, http://www.graphviz.org/, link tested January 2017.

[11] Beazley, D., PLY (Python Lex-Yacc), http://www.dabeaz.com/ply/ , link verified January 2017.

[12] D. Parson, D. Schwesinger and T. Steele,"Using Jython to Prototype and Extend Java-based Systems," *Proceedings of the 26th Annual Spring Conference of the Pennsylvania Computer and Information Science Educators* (PACISE), Shippensburg University, Shippensburg, PA, April 8-9, 2011.

[13] The Jython Project, http://www.jython.org/, link verified January 2017.

[14] Java Serialization, https://www.tutorialspoint.com/java/java_serialization.htm, link verified January 2017.

[15] Lucidchart, https://www.lucidchart.com/.

[16] PLANTUML, http://plantuml.com/, link verified January 2017.