Documenting Software Systems with Views V: Towards Visual Documentation of Design Patterns as an Aid to Program Understanding

Tim Trese
Department of Computer Sciences
Florida Institute of Technology
ttrese@fit.edu

Scott Tilley
Department of Computer Sciences
Florida Institute of Technology
stilley@cs.fit.edu

ABSTRACT
Cognitive science research indicates that a system is more readily understood when it is presented at progressive levels of decomposition, exposing increasing amounts of detail. One logical level of detail would present a system in terms of its implemented design patterns. However, to date, no entirely satisfactory method of documentation has been devised for explicating a software system as a set of design patterns. This paper discusses the challenges inherent in visualizing a software system as a set of design patterns, reviews the progress of another current effort, and describes a UML-compliant enhanced class-participation diagram as one possible solution.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *documentation*.

General Terms
Documentation, Human Factors, Standardization

Keywords
design patterns, documentation, program understanding

1. INTRODUCTION
It is commonly accepted that maintenance activities can account for as much as 80% of the total lifecycle cost of a software system. Frequently, the programmers conducting maintenance were not involved with the original development of the system and therefore possess a limited understanding of the system's design. To maintain the software in a disciplined, controlled manner, the first challenge such programmers face is one of program understanding: a learning process that involves, among other tasks, constructing a cognitive mapping from the functional requirements of a system in the application domain to the design and implementation of the system software at various levels of detail. A frequently cited goal of reverse engineering [3] activities such as program redocumentation [8] is to develop information products that facilitate this task.

SIGDOC'07, October 22–24, 2007, El Paso, Texas, USA.

In such redocumentation information products, a software system's design is generally explicated at several levels of detail, including the overall system, its component functional subsystems, and the source code itself. At an intermediate level between the functional subsystem and source code, object-oriented software can be described in terms of design patterns: cooperative assemblies of classes that collectively implement the solution to a low-level design problem in a well-known way [6]. Recognizing an implementation of a familiar design pattern might help programmers to more quickly and accurately acquire an understanding of the application. The recognized design pattern might also become a useful operand in further cognitive operations that the programmer performs to make correct decisions during system maintenance. If either of these statements is true, treatment of implemented design patterns adds value to redocumentation.

One of the most common strategies for documenting a system's static structure is with Unified Modeling Language (UML) class diagrams. Similar diagrams are also conventionally used to catalog design patterns. This paper argues that system redocumentation in which UML class diagrams are organized primarily for the optimal display of design patterns has several distinct advantages:

- Such documentation fosters program understanding by explicitly exposing implemented design patterns.
- Organization of UML class diagrams around design patterns using the visual design strategies proposed makes UML class diagrams semantically richer and more readable than traditional, comprehensive UML class diagrams.
- A CASE tool can semi-automatically generate the described UML class diagrams with minimal resort on the part of the documenting software engineer to a graphical UML design utility.

The next section of the paper discusses the importance of design pattern documentation as an aid to program understanding. Section 3 explains why it may be advantageous in documentation to represent design patterns using a repeated graphical template. Section 4 describes visual design strategies intended to augment explication of system architecture as a set of design patterns. Finally, Section 5 summarizes the paper and outlines possible avenues of further work, such as the research remaining to produce empirical validation of the ideas presented herein.

2. DESIGN PATTERNS IN DOCUMENTATION

Design patterns are recognized as valuable aids in software engineering. They provide language-independent solutions to common design problems. This section discusses how the same approach can be used in terms of documentation patterns to aid program understanding.

2.1 Advantages of Design Pattern Documentation

Design patterns aid in the program understanding process, both from the top-down, system decomposition, and the bottom-up, detail aggregation, perspectives.

2.1.1 Top-Down: Decomposition of a System into Familiar Functional Assemblies

Empirical evidence supports the fairly intuitive proposition that design pattern documentation is an aid to program understanding. Prechelt et al. have run controlled experiments demonstrating that "[pattern comment lines] in [source code headers] may considerably reduce the time required for a program change or may help improve the quality of the change" [11]. It seems reasonable to conclude from these findings that inserting pattern comment lines as part of a redocumentation effort would facilitate program understanding, but there are some problems with doing so. For example, documenting classes' participation in an external design pattern within a class violates the tenet of class encapsulation. Moreover, redocumenting patterns in source code is intrusive into the standing code base, which may not be practicable or desirable under all circumstances. Both of these problems are overcome by capturing design pattern information in some external document.

2.1.2 Bottom-Up: Chunking of Classes into Meaningful Groups

Studies in program comprehension indicate that when trying to understand a program, programmers frequently use a chunking strategy, a learning mechanism leading to the acquisition of long-term memory structures that can be used as units of perception and meaning [17]. This strategy implies a need for the programmer to reduce the number of cognitive elements to a manageable size. Industrial-size software systems can comprise thousands of classes. While a system design can often be functionally decomposed into a cognitively-manageable number of more-or-less independent subsystems, each of these subsystems might itself consist of dozens up to perhaps hundreds of classes. So, even after this first level of decomposition, these subsystems pose a formidable challenge to program understanding by dint of sheer numbers.

The next level of functional decomposition is into design patterns: so-called "architecture in-the-small." Each design pattern provides a grouping of several classes into a discrete, functionally significant element of the implementation domain. The number of design patterns implemented in a subsystem can vary widely, but decomposition into such groupings provides relief to program comprehension for two reasons. First, the design pattern itself serves as a discrete cognitive operand for tasks involving low-level architecture. Second, the number of design pattern elements in a subsystem at least helps to approach something cognitively manageable.

A design pattern can be further decomposed into several participants, each of which comprises either a class or a cluster of classes that inherit from another participant. The number of participants in a design pattern is usually a very manageable number: a census of the patterns catalogued in the classic *Design Patterns* text [6] shows a range of from one to five participants, with the average being a little more than three participants in any given pattern.

2.2 Challenges to Documenting a System as a Set of Design Patterns

Redocumentation of low-level architecture and design patterns admits of several difficulties: those related to identifying of design patterns in software, those of aberrant design in the system under study, and those relating to the nature of how design patterns function within the system.

2.2.1 Lack of Automated Pattern Identification

Automated identification of design patterns in an architecture remains an elusive goal. While this is fertile ground for research (e.g. [4], [10]) the state of the art is, at best, semi-automated. This necessitates that a human re-documenter with a strong design pattern vocabulary also cull the system for patterns and manually document them as they are found.

2.2.2 Inapplicability of Patterns

Exacerbating the preceding difficulty is that not all software designs make optimal use design patterns or implement them in a standard way. The absence of design patterns, non-standard implementations, and the existence of so-called anti-patterns [2] present a challenge to the decomposition of a low-level architecture into its component design patterns.

2.2.3 Limited Pattern Vocabulary of Documenter and Maintainer

Given the state of the art in automated pattern identification, it remains incumbent upon the documenter to have working knowledge of design patterns to recognize classes and members within a subsystem so that he can tag them appropriately. Given the scope of the subject, it is unlikely that any documenter is aware of all known patterns or even all of the variations on those patterns with which he is familiar. Further, knowledge of design patterns among programmers using this documentation to acquire program understanding can vary widely, and merely recognizing that a class participates in a particular pattern is not sufficient without some understanding of the significance of the pattern.

To accommodate both documentation producers and consumers, a CASE tool that enables the sort of documentation contemplated here must provide a design pattern catalog. To facilitate the production of UML class diagrams with pattern information, the CASE tool could provide a mechanism that enables the documenter to the rapidly map program classes and members onto elements of a selected structure from the catalog.

2.2.4 Class Participation Is Not an Equivalence Relation

Most relevant to this paper is the fact that design patterns do not generally permit a "neat" decomposition of a subsystem into lower-level elements in the same way that a system can be broken down into its constituent subsystems or that a design pattern decomposes into participants. Some classes will participate in more than one design pattern, and some classes, hereafter referred to as "auxiliary" classes, do not participate in any known design patterns at all. In other words, class participation in a design pattern is not an equivalence relation; design patterns do not strictly partition the set of classes in a subsystem. This presents a special challenge to redocumentation of low-level architecture that will be discussed in subsequent sections of this paper.

3. VISUAL REPRESENTATION OF DESIGN PATTERN STRUCTURES IN EXPLICATING DESIGN

This section explains why it may be advantageous in documentation to represent design patterns using a repeated graphical template.

3.1 Problems Addressed

Using standard, widely accepted visual representations of a design pattern might address issues in both program understanding and efficient development of documentation.

3.1.1 Canonical Representations Leverage Visual Convention and Parallelism

The most recognizable visual representation of a design pattern is the simple UML or quasi-UML diagram that represents its structure. The design pattern structural diagram as it is commonly catalogued will here be referred to as its "canonical" representation. As a programmer's vocabulary of design patterns grows, so too does their familiarity with these canonical representations; in effect, the convention of the canonically-represented design pattern becomes a familiar and recognizable "phrase" in the visual language.

Tufte describes the value of parallelism in graphical display of information as follows:

> "Parallelism connects visual elements. Connections are built among images by position, orientation, overlap, synchronization, and similarities in content. Parallelism grows from a common viewpoint that relates like to like. Congruity of structure across multiple images gives the eye a context for assessing data variation. Parallelism is not simply a matter of design arrangements for the perceiving mind itself actively works to detect and indeed to generate links clusters and matches among assorted visual elements" [16].

This paper suggests that displaying an implementation of a design pattern with participating classes in the same position and orientation as appears in the canonical representation of the design pattern, and showing only selected information specified by the canonical representation, leverages parallelism and established visual convention, thereby facilitating more rapid program understanding.

3.1.2 Participants Always in Respective Locations

While further empirical evidence is demanded, there are compelling reasons why one would expect this claim to be true. In the canonical representation of a design pattern, the visual structure assigns each participant a relative location in 2D space. If one is attempting to identify the participants in an implemented design pattern, having each participant in the same position relative to other participants provides an immediate visual queue as to its function in the pattern. It may well prove that a programmer sufficiently fluent with a particular pattern and its canonical representation could immediately identify the participants in an implemented pattern by position alone.

3.1.3 Templates Enable Semi-Automated Documentation

Another advantage to displaying implemented design patterns in their canonical representation is that it semi-automates the process of creating readable UML class diagrams. A relatively simple tool could be devised that would enable a re-documenter to map elements, classes, and relevant class members of an implementation onto the set of participants in a selected design pattern, and have the implementation automatically displayed in a UML class diagram in the pattern's canonical representation.

3.1.4 Addressing the Challenge of Readable UML Class Diagrams

Redocumentation makes ample use of UML class diagrams as a software visualization aid to program understanding. As Tilley and Huang point out, it is, in a sense, a standard. However, in presenting their findings about the efficacy of UML as an aid to program understanding, they identify among UML's shortcomings that "for complex UML [class] diagrams, which can have many dozens of artifacts and an equally large number of relationship arcs, the problem [of graph layout and detail visibility] is particularly acute" [14].

Approaches to solving the problems of UML class diagram layout that conform to the syntax of the language can be broken into three general categories: applying additional constraints to UML, applying graphical innovations that augment UML, and applying selectivity to the information depicted in a given diagram.

Constraining UML is the purpose of the style guidelines asserted by Ambler in *The Elements of UML Style* [1]. The establishment of a hierarchy of aesthetic preferences to enhance readability of UML, and the subsequent assessment of automated UML layout features in CASE tools according to these standards, has received considerable attention (e.g. [5], [13]). An example of the graphical innovations suggested for UML class diagrams is the use of color to highlight class hierarchies and generalizations [7].

Selectivity of information is endorsed by Ambler in his general diagramming guidelines: "show only what you have to show" and "reorganize large diagrams into several smaller ones."

3.2 Other Approaches and Challenges

The standard UML method for representing design patterns in a class diagram presents readability issues. Some new kind of

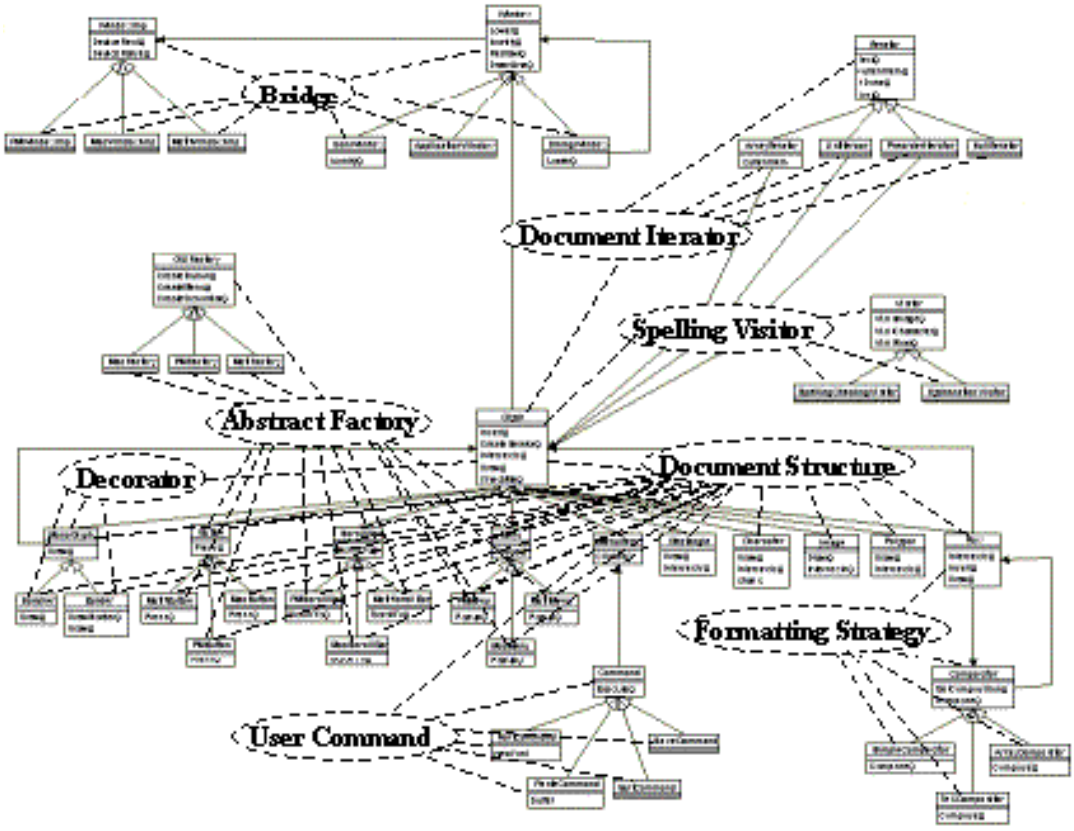

Figure 1: Design Patterns UML Standard Notation [12]

pattern diagram may prove a good solution, but any such innovation must address the equivalence relation problem inherent in design patterns.

3.2.1 No Standard "Pattern Diagram"

Using the Lexi program documented in [6] as an example, Schauer and Keller graphically articulate the problems of using a standard UML class diagram with the language-prescribed design pattern markup as shown in Figure 1 [12]. Although the depiction of classes in relative proximity and location approximates the canonical structure representation of design patterns, any benefit is obscured by the visual noise generated when the standard design pattern notation is added.

The authors present as an alternative the "pattern collaboration diagram" shown in Figure 2, a featured product of the tool that they are developing. This diagram achieves readability by depicting design patterns in place of classes in a UML class diagram-like format. Although the diagram shows limited information about participating classes, when displayed in the tool, the reader can select a pattern and expand it to see its constituent classes. After further research and scrutiny by a broad-based consortium like OMG, innovations like Figure 2 may well provide a standard visual language for showing design pattern implementations in an architecture.

Note, however, that some semantic problems need to be rigorously addressed if the pattern collaboration diagram is to become a standard. In the Lexi application example used in [6], the abstract class Glyph is the Component participant in a Composite pattern and also the Component participant in a Decorator pattern. The relation between the Composite pattern and the Decorator pattern is not, as depicted in Figure 2, simultaneous aggregation and generalization; patterns are not generally said to aggregate or generalize other patterns. Rather, if the pattern collaboration diagram is to precisely depict relations between atomic design patterns, perhaps the arc between them should simply indicate that the Composite and Decorator patterns share a class in common.

Further, if all of the patterns in Figure 2 are collapsed to indicate atomic design patterns, the only apparent mapping from the design patterns down to the class-implementation level of decomposition is the incomplete lists of participating classes identified in the pattern boxes. Exactly how and why these classes are identified to the exclusion of other participants is unclear, but this approach obscures the reader's cognitive link from design patterns down to the lowest level of the implementation domain.

3.2.2 Multiple Design Patterns in Same Visualization

One of the principal challenges to be met in visually representing a subsystem as a set of design patterns in a UML class diagram is overlap. Because the same class can participate in multiple design patterns, rendering design patterns in their canonical structures means that the design patterns overlap, with consequent readability problems. The alternative proposed here to overcome this difficulty is to represent the same class multiple times in the

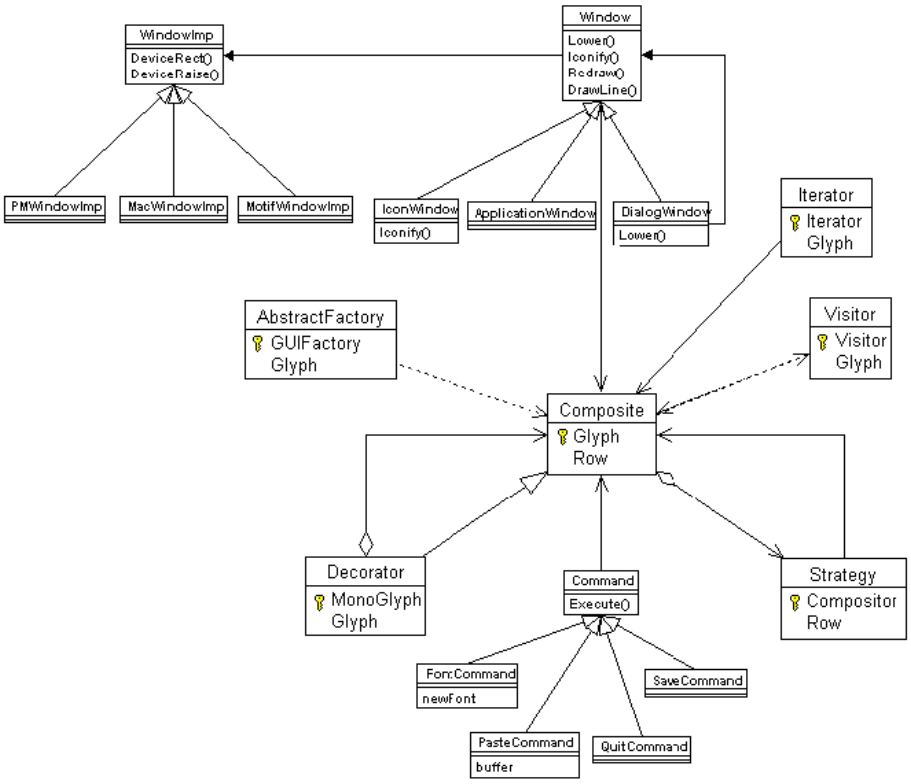

Figure 2: Pattern Collaboration Diagram [12]

same diagram, and provide graphical cues to the reader that the same class is present several times in the diagram.

3.2.3 Auxiliary Classes and Members

As noted previously, there are auxiliary classes in a subsystem that do not participate in any known design patterns. In an explication of a subsystem as a set of design patterns, there is no assigned place for such classes. Moreover, even in classes that participate in multiple design patterns, there will likely be auxiliary members of these classes: behaviors and attributes that serve internal purposes and are not relevant to any of the patterns in which the class participates.

In order to achieve a complete representation of a subsystem as a set of design patterns, this paper proposes that auxiliary classes be grafted on to a canonical structure where they make most sense. In order to exhaustively explicate a class, there will have to be at least one additional place-holder representation of that class to capture its auxiliary members.

4. UML-COMPLIANT ENHANCEMENTS FOR DEPICTING CLASS-PARTICIPATION USING CANONICAL DESIGN PATTERN REPRESENTATIONS

The goal of this paper is to propose requirements for the display of highly readable UML-class diagrams that facilitate cognitive chunking and fully leverage the advantages of displaying design patterns in their canonical representations, while surmounting some of the difficulties noted in the preceding paragraphs. What remains is to identify and provide examples for appropriate visual design strategies in such class diagrams.

4.1 Interrelating Pattern Structures as a Set of Class-Participation Diagrams

To facilitate chunking as a cognitive strategy for subsystem comprehension, we propose that the programmer visualize the subsystem in a series of UML class diagrams. Each of these class diagrams depicts the canonical representations of all the patterns in which a selected class participates, here called "class-participation diagrams." The class-participation diagram also contains a non-pattern depiction of the selected class, displaying its members that do not participate in patterns, and it may also depict auxiliary classes identified by the documenter that are in some way coupled to the selected class. These auxiliary classes may be connected by standard UML arcs to the non-pattern depiction of the selected class or to a depiction of the selected class as a pattern-participant, whichever the documenter deems most relevant.

4.2 Figure-Ground Visual Design Strategy

Visual emphasis of selected elements provides the reader with focal points that are the cognitive entry points into a visual field and identify what is most important [9]. Our diagramming

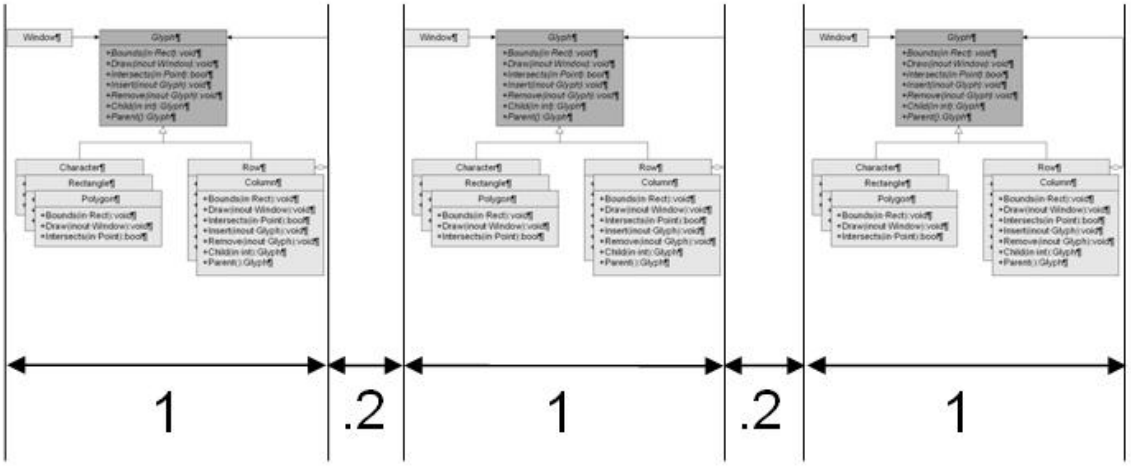

Figure 3: Spacing between Patterns

approach implies three gradations of emphasis for depicting classes: the primary selected class that participates in all design patterns on the class-participation diagram, secondary classes that are co-participants in the design patterns depicted, and tertiary auxiliary classes that the documenter has chosen to depict with the primary class. In the class-participation diagram, each of these gradations is assigned a respective degree of figure-ground contrast, achieved by monochromatic shading of the class boxes in the diagram. According to Tufte, only minimal contrasts between elements and ground are necessary to produce a visual hierarchy [16]. Consequently, we have chosen 30%, 10%, and 0% tints to provide visual gradation in Figure 4. Note that these are easily distinguishable gradations that do not interfere with reading the foreground black type.

4.3 Visual Grouping Design Strategies

Visual grouping strategies organize a visual field into units and subunits, "[enabling] readers to sort through the parts of a document more efficiently [and creating] visual cohesion" [9]. In essence, grouping strategies facilitate navigability and further cognitive chunking. Several grouping strategies are employed in our approach.

4.3.1 Spatial Nearness

The primary application of spatial nearness grouping in the class-participation diagram is accomplished by the canonical representations of the design patterns themselves, which keep the participating classes in close spatial proximity. Adequate white space margins around each pattern (about 20% of the width of the canonical representations in Figure 3) maintain a distinct boundary between the patterns so that even at a cursory glance it is obvious in which pattern any depicted class is a participant.

4.3.2 Division

A second grouping strategy is applied by dividing the patterns depicted into several distinct groups: three group patterns of the purposes assigned by [6] (creational, structural, and behavioral), one group for the non-pattern depiction of the primary class, and one group for anti-patterns. In Figure 5, this division is achieved using relatively heavyweight (4 pt.) boundary lines that are easily distinguishable from the (1 pt.) relation arcs and class rectangle borders that are typical of most UML class diagrams.

Note that over the set of class-participation diagrams, these five divisions are maintained in the same location relative to one another, as shown in Figure 5. Consistent application of this visual convention enables the reader to readily navigate to the three purpose divisions, the non-pattern division, and the anti-pattern division.

4.3.3 Rows

Within each division described above, patterns are arrayed in rows in Figure 5. This enables further ease of navigation, as the user can scan horizontally for a design pattern of interest within its division. This has a secondary beneficial effect of arranging the complete class-participation diagram in a rectangular, rather than horizontally or vertically linear, layout, suitable for printing or on-screen display at some arbitrary degree of magnification that fits the presentation medium.

5. SUMMARY

Good documentation of software is clear, concise, and provides readable coverage of the system's implemented design patterns. This paper described the underlying rationale and outlined requirements for a documentation suite that includes class-participation diagrams. UML-compliant enhancements for depicting such diagrams using canonical design pattern representations were presented.

5.1 Future Work

There are several avenues for possible future work in this area. Two that will receive close attention are enhancements to an existing documentation maturity model based on the ideas proposed in this paper, and empirical validation of the diagram enhancement techniques.

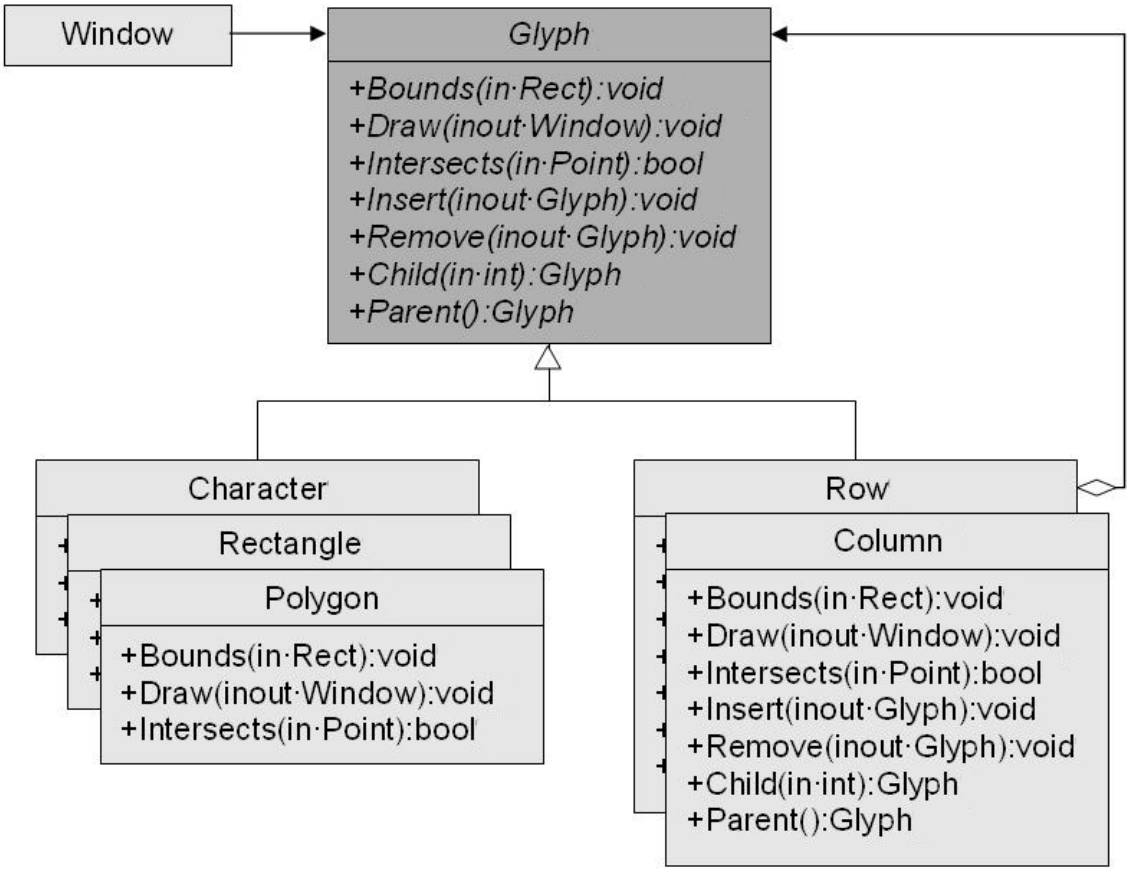

Figure 4: Strategy Pattern from Lexi with Shading

5.1.1 Enhancing the Documentation Maturity Model

In [15], Tilley and Huang propose a Documentation Maturity Model (DMM) that outlines a set of heuristic key product attributes (KPAs) against which the quality of redocumentation can be evaluated. One of these KPAs is granularity: the ascending level of abstraction to which a system has been documented. Another KPA is graphical format. Documentation conforming to the requirements presented in the present paper is demonstrably at the Design Patterns (DMM product level 2) of granularity and Static and Standardized (DMM product level 2) of graphical format. We believe that as further enhancements are suggested by findings of the research outlined above, this could drive further product innovations that enable higher maturity levels of re-documentation and provide products of greater value to the programmer engaged in acquiring program understanding.

5.1.2 Empirical Validation

Considerable work remains to empirically validate the ideas presented by answering the following questions:

(a) Does depicting an implemented design pattern in its canonical representation really make any difference?

An experiment must be devised for testing the hypothesis that readers will more quickly and easily understand a design pattern implementation if it is documented visually, using a familiar congruent canonical representation. An experimental task asked of the subjects should simulate an industrial program understanding task.

(b) Does the documentation envisioned facilitate software maintenance better than alternatives?

A second experiment should seek to quantify how much more efficiently programmers can maintain a system when that system is fully documented with class-participation diagrams, as opposed to some other control documentation representative of current industry standards. This will provide baseline data on the value of such documentation to the users.

(c) Can documenters generate class-participation diagram documentation as easily as supposed?

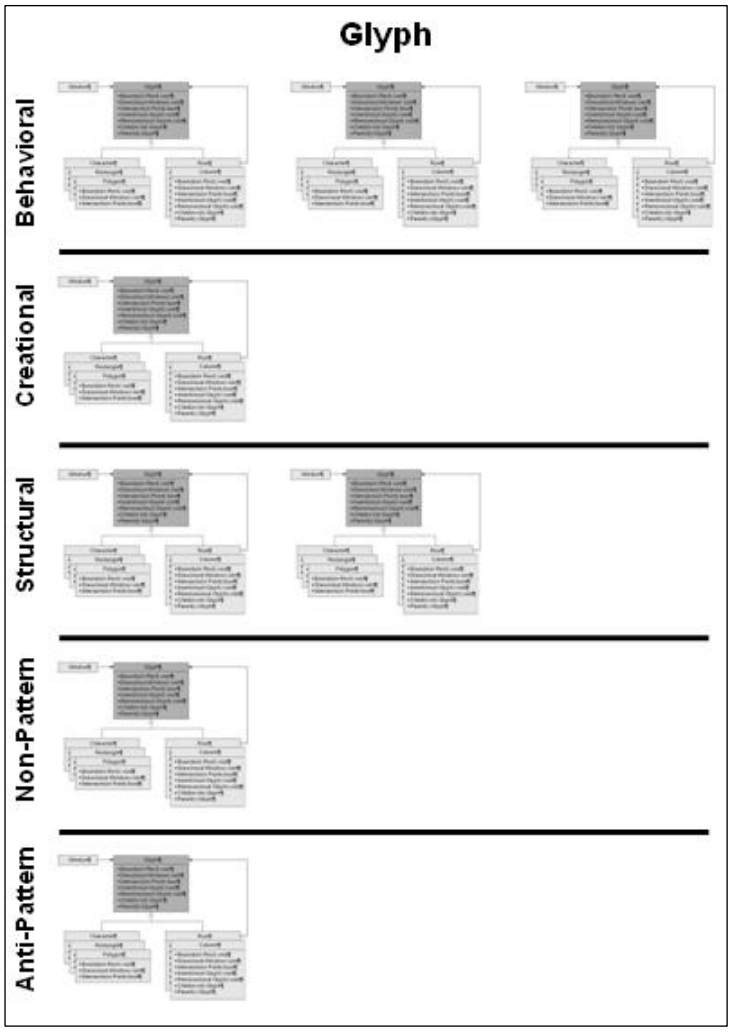

Figure 5: Layout of Class Design Pattern Information

To determine if the proposed documentation presents a winning value proposition, it is also necessary to estimate the costs of producing class-participation diagrams relative to current industry-standard redocumentation. This should be experimentally determined by developing a prototype CASE tool that provides the capabilities to document a system and comparing this with time required to produce alternative documentation. One challenge to the design of this experiment is creating a standard comparable to the one developed here, but for the alternative documentation.

REFERENCES

[1] Ambler, S. *The Elements of UML 2.0 Style*. New York, NY: Cambridge University Press, 2005.

[2] Brown, W., Malveau, R., McCormick, H., and Mowbray, T. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. New York: Wiley & Sons, 1998. pp. 7-8

[3] Chikofsy, E.; and Cross, J. "Reverse Engineering and Design Recovery: A Taxonomy." *IEEE Software* 7(1):13-17, January 1990.

[4] Costagliola, G., De Lucia, A. Deufemia, V., Gravino, C. and Risi, M. "Case Studies of Visual Language Based Design Patterns Recovery" *Proceedings of the Conference on Software Maintenance and Reengineering* (CSMR'06) IEEE, 2006.

[5] Eichelberger, H. "Nice Diagrams Admit Good Design?" In SoftVis '03: Proceedings of the 2003 Symposium on Software Visualization, pp. 159-168. New York: ACM Press, 2003.

[6] Gamma, E and R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass: Addison-Wesley, 1995.

[7] Gutwenger, C., Jünger, M., Klein, K., Kupke, J., Leipert, S. and Mutzel, P. "A New Approach for Visualizing UML class Diagrams." ACM Symposium on Software Visualization. San Diego, CA, 2003 pp. 179-188.

[8] Huang, S. *An Integrated Approach to Program Redocumentation*. Ph.D. Dissertation, University of California, Riverside, 2004.

[9] Kostelnick, C. and Roberts, D. *Designing Visual Language: Strategies for Professional Communicators*. Needham Heights, MA: Allyn & Bacon, 1998.

[10] Mak, J., Choy, C., and Lun, D. "Precise Modeling of Design Patterns in UML." *Proceedings of the 26th International Conference on Software Engineering* (ICSE'04)

[11] Prechelt, L.; B. Unger-Lamprecht; M. Philippsen; and W. Tichy. "Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance." *IEEE Transactions on Software Engineering*, Vol. 28, No. 6, June 2002. pp. 595-606

[12] Schauer, R. and Keller, R. "Pattern visualization for software comprehension." *Proceedings of the 6th International Workshop on Program Comprehension* (IWPC '98: June 24-26, 1998; Ischia, Italy), pp. 4 – 12. Los Alamitos, CA: IEEE Computer Society Press, 1998.

[13] Sun, D. and Wong, K. "On Evaluating the Layout of UML Class Diagrams for Program Comprehension." *Proceedings of the 13th International Workshop on Program Comprehension* (IWPC 2005: May 15-16, 2006; St. Louis, MO), pp. 317 – 326. Los Alamitos, CA: IEEE Computer Society Press, 2005.

[14] Tilley, S. and S. Huang. "A Qualitative Assessment of the Efficacy of UML Diagrams as a Form of Graphical Documentation in Aiding Program Understanding." *Proceedings of the 21st Annual International Conference on Design of Communication* (SIGDOC 2003: October 12-15, 2003; San Francisco, CA), pp. 184-191. ACM Press: New York, NY, 2003.

[15] Tilley, S. and S. Huang. "Towards a Documentation Maturity Model." *Proceedings of the 21st Annual International Conference on Design of Communication* (SIGDOC 2003: October 12-15, 2003; San Francisco, CA), pp. 93-99. ACM Press: New York, NY, 2003.

[16] Tufte, Edward R. *Visual Explanations: Images and Quantities, Evidence and Narrative*. Cheshire, CT: Graphics Press, 1997.

[17] von Mayerhauser, A. and Vans, A. "Program Understanding during Large Scale Debugging of Software." *Papers Presented at the Seventh Workshop on Empirical Studies of Programmers*. New York: ACM Press, 1997. pp. 157-179.