# Pattern Puzzle: A Metaphor for Visualizing Software Complexity Measures

**Adam Ghandar, A.S.M Sajeev, Xiaodi Huang**

School of Computer Science
University of New England
Armidale, New South Wales, Australia

{aghandar,sajeev, xhuang}@cs.une.edu.au

## Abstract

Software systems have become increasingly complex over the years. Complexity metrics measures software complexity using real numbers. It is, however, hard to gain insight into different complexities by looking at these numbers. In this paper, we present a software complexity metaphor that uses a jigsaw puzzle. In particular, each component of a software system is modelled as a piece of a jigsaw puzzle. The problem complexity is modelled as a pattern on the surface of the piece, and the interconnection complexity as connectors between puzzle pieces. We demonstrate the benefits of this approach using case studies of the complexity measures of a real software system.

Keywords: Visualization, User Interface, Complexity Measures, Software Engineering

## 1 Introduction

Computer software has increased in complexity over the years. Software plays an important role in human life, from controlling advanced defense systems and power stations to remote education and games. Designing and maintaining complex software places an assortment of demands on all the actors in the design process. Managers must allocate human and material resources efficiently where they are needed; programmers and analysts need to form an understanding of the design of the software system. It is often necessary to present information about software systems to actors from different professional backgrounds.

We propose a method called the pattern puzzle metaphor that visualizes the measures of software complexity. The idea is based on the concept of a jig-saw puzzle – a set of pieces that fit together to create a larger picture. We implemented this method in a prototype to demonstrate the benefits of the metaphor and its use. In essence the pattern puzzle allows a user to form an understanding of at least two dimensions of complexity; that is, how they are related to each other, and how the complexity is distributed throughout a software system.



## 2 Complexity Measures

### 2.1 Problem Complexity

Some problem domains are inherently more complex than others. For instance, a software system for launching antiballistic missile in a defense establishment is likely to be much more complex than a software system that teaches the English alphabet. The Problem Complexity indicates complexities that arise from the problem domain. Furthermore, it is common that within the same application, some components have higher problem complexity than others. For example, in a search engine that is able to accept natural language questions, the processing component which analyses the natural language input is likely to be more complex than the user-interface component which displays the results to the user. The problem complexity of each component or a group of components (say, a package) is relevant to that of the whole system. As a result, managers may want to allocate additional resources and/or set up specific quality assurance procedures for components that are highly complex. However, as the number of components becomes large, it wouldn't be easy to get a good "picture" of the distribution of problem complexity among the different parts of the system or to locate those components that need a closer look. This is where visualization helps.

### 2.2 Design Complexity

Another class of complexity is the one arising from the way in which a software system is designed. For the same application, poorly designed software may exhibit higher complexity than a better designed one. Measures such as coupling, cohesion, weighted methods per classes, depth

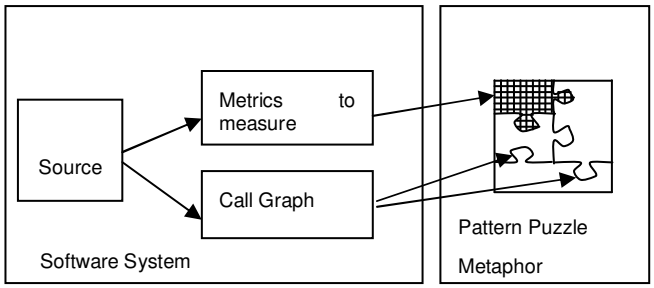



**Figure 1. Overview of the Pattern Puzzle Metaphor: the call graph is the source of the pieces positions and their connections.**

of inheritance tree etc. (Chidamber and Kemerer, 1994) are used in the literature to measure design complexity. Design complexity has implications both in developing reliable software as well as in maintenance of the system over its life cycle. Traditionally, graphs such as call-graphs (where nodes represent components and edges represent calls from one component to another) have been used to visualize aspects of design complexity.

## 3 Pattern Puzzle Metaphor

The Pattern Puzzle metaphor provides an integrated approach to visualizing complexity of software systems. It is based on the concept of a jig-saw puzzle – a set of pieces that fit together to create a larger picture. This larger picture is, in the same way that a real jig-saw puzzle is, the sum of its parts which are the pieces.

The metaphor has two main components: the *pieces* together with their individual pictures and the *joins* between the pieces along with the locations of pieces (see Figure 1). With reference to the concept of planar and retinal variables (Bertin, 1983), the dimensions of the pieces are planar and the patterns are a combination of several retinal variables.

The effectiveness of the metaphor is amplified by the careful choice of the surface pattern so that universal symbols (Fromm, 1976) are generated. If the pattern contains an intrinsic relationship with the attribute being visualized, then the Pattern Puzzle becomes a lucid and expressive way of looking at the software system.

### 3.1 Patterns

Surface patterns are a part of each puzzle piece and a target mapping for one or more attributes of the underlying software system. The patterns allow the viewing of at least one further attribute of a system in the same way that, say, the color would. However a pattern may be thought of as the amalgamation of a subset six retinal variables mentioned in (Bertin, 1983) (color, brightness, etc) in a coherent form where each variable complements the other. This means that a pattern, in some senses, can impart meaning by utilizing both the variables themselves and the relationships between them. Compared with approaches that use a single retinal variable such as the color to represent an additional attribute, patterns are a complimentary extension.

### 3.2 Connections

We elected to represent program complexity using an interesting class of patterns. They are symmetrical geometric shapes that are able to be generated in a computer graphics form using variables according to a method (Kaplan and Salesin, 2004). They are members of a class of designs termed Islamic Star Patterns. Figures 2 depicts a series of six pointed rosettes generated with the assistance of Kaplan's Taprats (Kaplan, 2000) library, and are a common motif used historically.

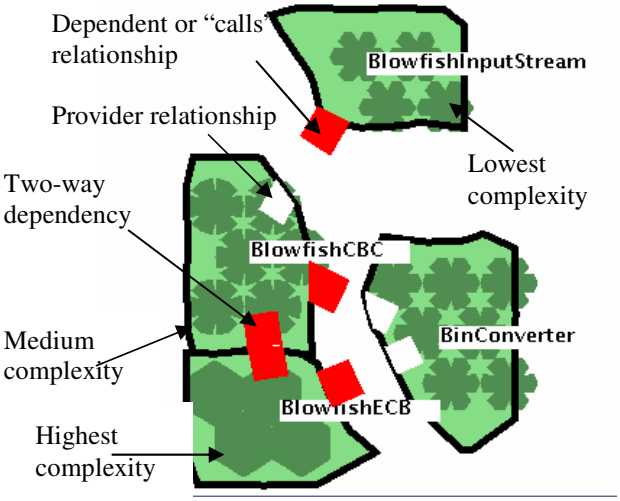



**Figure 2. The meaning of symbols used in the implementation of the Pattern Puzzle metaphor.**

The Pattern Puzzle pieces may be connected by puzzle joins to symbolize relational attributes in a software system. In fact, a number of problems arise in traditional graph visualization techniques when they are used to visualize software systems. The Pattern Puzzle metaphor attempts to lessen the confusion that arises from these problems by focusing on the puzzle pieces and by incorporating visual information about both the piece and its relationship to other pieces.

Puzzle joins are represented by protrusions of one piece's area into another. However, when creating a puzzle from a graph it is likely that a graph is not planar. In this case it is necessary to insert additional puzzle pieces. These pieces are drawn grey and the user is given a visual cue that the interconnection complexity is high.

A user is able to form an idea of the number of relationships in the data by briefly looking at the puzzle. This aspect of the visualization is enhanced by selecting a bright color for the links and by ensuring that they are of a reasonable size. Puzzles can be compared quickly with respect to the extent of interconnection among the components. Greater numbers of links indicate a higher interconnection complexity in the underlying system.

### 3.3 Hierarchy and User Interaction

The metaphor incorporates a hierarchical depiction of software systems in two ways.

First, patterns, which represent complex components in a puzzle that represents non-atomic components, (for example classes in a Java program can be divided into methods) imply that a complex structure lies 'below' the piece. The second level hierarchy allows users to 'dig' into the puzzle by selecting the pieces and to replace it with a new one in order to represent the subcomponents of a component, which is represented by the piece in the original puzzle. It is also be possible to display several puzzles of different hierarchical levels simultaneously. The case study, illustrated in Figure 3, demonstrates the use of this method for visualizing hierarchy.

Users can also be encouraged to fit the puzzle pieces together themselves. This helps the users to build their mental maps of aspects of the software system incorporated in the visualization.

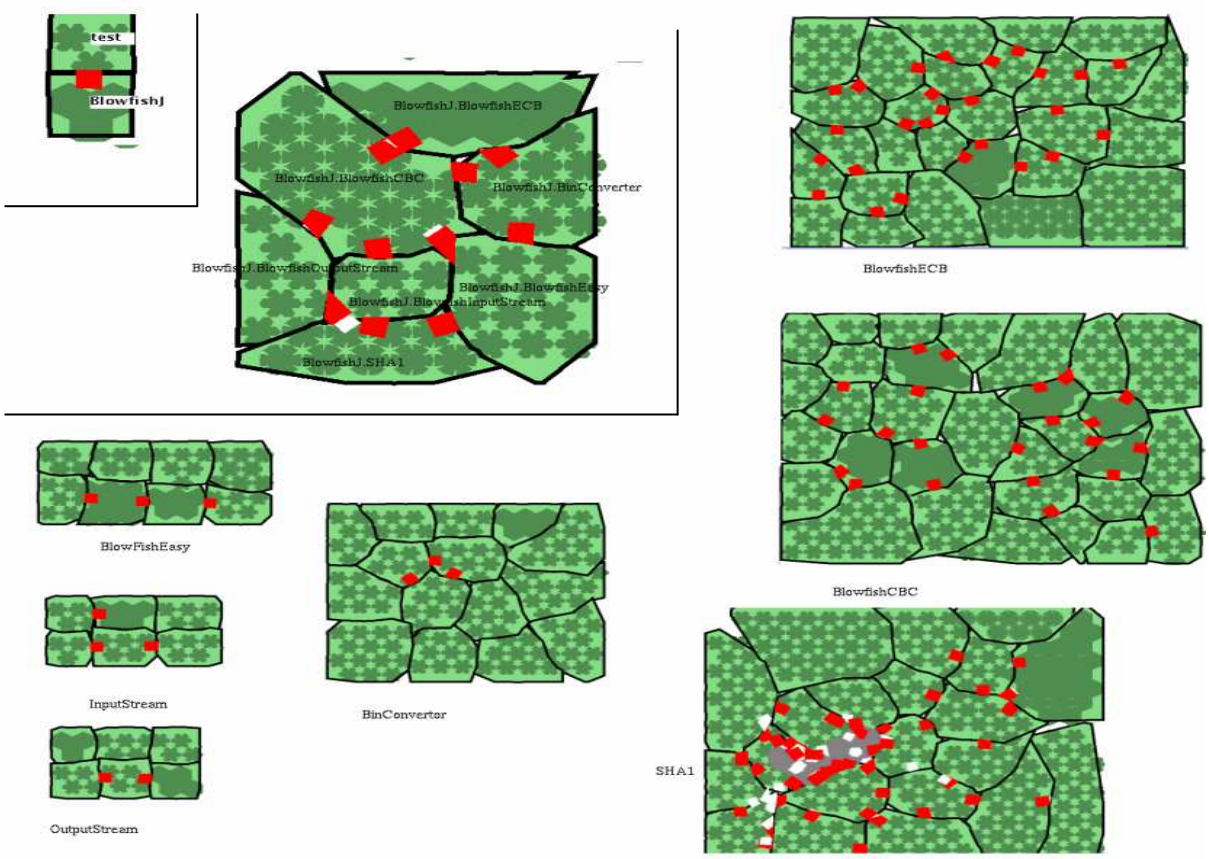



**Figure 3. Visualizations of the BlowfishJ library by the Pattern Puzzle. Two packages: BlowfishJ and test, and their sub-components in the library are shown. Puzzles representing the three levels of hierarchy (package, class and method) are drawn. Note that the patterns in each puzzle indicate the relative value of complexity to those of other pieces in the same puzzle, instead of the overall value. The SHA1 pattern puzzle (bottom right) illustrates high interconnection complexity the diagram visually implies that the component could need more resources than the others.**

## 3.4 Formal Description

In this Section, we formalize the problem of the Pattern Puzzle metaphor for visualizing software complexity measures.

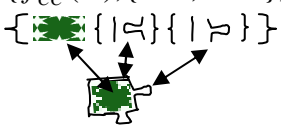
A puzzle piece is the basic element in the visual vocabulary of the metaphor denoted by $V$ = {Pattern Puzzle} = {Puzzle piece, Connector}. Each piece comprises of four elements: Puzzle piece = {surface picture, color, size, shape}. The relationships between Puzzle piece and Connector are represented as $\text{Connector} \subseteq \text{Puzzle piece} \times \text{Puzzle piece}$. The connection includes two types: protrusions or intrusions, namely Connector = {intrusion, extrusion}, both of which may be simultaneously present between two adjacent pieces.

In the design of the metaphor, we face three fundamental problems: (1) choose the software complexity measure $H$ and a visual vocabulary $V$; (2) calculate the complexity values associated with each components of software denoted by $Q = f_H(c)$; and (3) then find their mappings. That is, $D: Q \to V$. In particular, the mappings in our system include the followings:

- The complexity of software component Puzzle Piece, namely $f_M(c_{ij}) \to$ Puzzle piece where $j$ is a constant integer representing the structure level of components;
- $f_{CC}(m) \to$ Pattern, which maps the complexity of class method $m$ calculated by the cyclomatic complexity (CC) metric, into the Pattern;
- $R_j^G \to$ Connector, which maps the relationships between components into the connections of puzzle pieces.

In addition, we illustrate the relationships between components $R_j^G$. For instance, in the method level, we have {Cyclomatic complexity, dependant upon, depended upon} {Pattern, intrusion, extrusion}. As an example, consider the following mapping:

$\{f_{CC}(m), \{\text{true}, \text{false}\}, \{\text{true}, \text{false}\}\} \to$



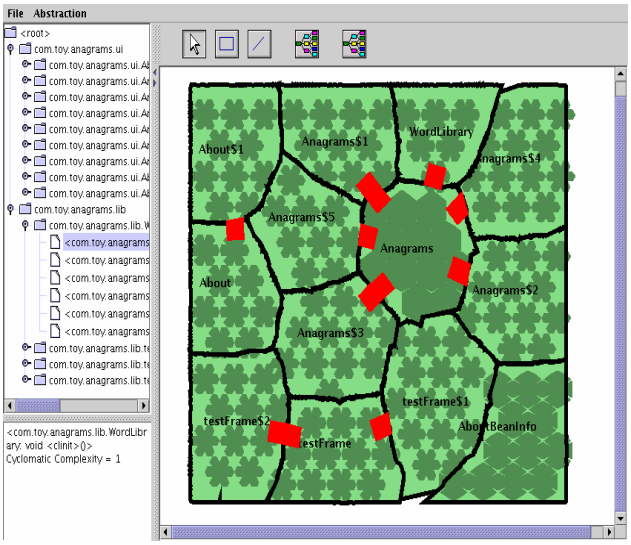



**Figure 4. The user interface of the implementation of the Pattern Puzzle Metaphor prototype.**

The true and false elements refer to whether or not a component is dependant on another specific component in a software system.

## 4 Case Study

We developed a tool which produces pattern puzzle representations of Java source files (see Figure 4). The tool is able to show three levels of hierarchy based on the package, class and method of a Java file.

We demonstrate the metaphor with the Blowfishj library (Hahn and Goeschl, 2004) which is of a reasonable size and exhibits large variations in complexity across components. It comprises 2 packages, 10 classes, and 2688 non-commented lines of code. Several classes have high interconnection complexity, and a number of methods have very high cyclomatic complexity (greater than 10). This is mainly because of the inherent problem complexity of the Blowfish algorithm (Schneier, 1993).

Figure 3 shows a visualization of the Blowfishj library in different levels using our Pattern Puzzle metaphor implementation. The three levels of hierarch are shown. Complexity is indicated by the pattern on the surface of the pieces. Highest complexity is implied by solid blocks of colors. Elegant rosettes imply simplicity. Levels of complexity are implied by gradations between these extremes (see Figure 2). Note that the patterns are meaningful only within each puzzle and not relative over the whole of Figure 3.

A closer inspection of the puzzles in Figure 3, for example, reveals further properties of the software system. The method level puzzles of BlowfishCBC and BlowfishECB show simpler dependency patterns (indicated by the joins) than SHA1 that has fewer methods but with comparable or higher complexity. The method puzzle of SHA1 reveals a high level of interconnection complexity and a single method whose complexity dwarfs the others in the class. This would imply that the allocation of more resources (time, skills) to the implementation of the SHA1 class is necessary. It may also signify that the design of this class should be looked at closely.

Many aspects of software can be perceived from the puzzle representation in addition to complexity. For example afferent and efferent couplings. At the package level the number of extrusions is equivalent to the number of efferent couplings and indicates packages independence. Conversely the number of intrusions indicates afferent couplings and indicates the package provides services to other components.

## 5 Conclusion

In this paper, we have presented our new approach called Pattern Puzzle metaphor for visualizing software complexity measures. One of the most successful metaphors used in computing is the desktop metaphor for the interfaces of operating systems. This success lies partly in the effective use of symbolism and user interaction which enable the user to make intuitive associations between the operating systems and the desktop metaphor without a great deal of learning. In other words, the desktop metaphor is almost a universal symbol (Fromm, 1976) for computer users. The Pattern Puzzle metaphor attempts to fulfil these criteria in the area of software complexity visualization using the jig-saw puzzle symbol.

## ACKNOWLEDGEMENT

This research is supported by the Australian Research Council Discovery Grant, DP0209483.

## REFERENCES

Bertin, J. (1983): *Semiology of graphics,* University of Wisconsin Press.

Chidamber, S. R. and Kemerer, C. F. (1994): A Metrics Suite for Object-Oriented Design *IEEE Transactions on Software Engineering,* **20,** 476-493.

Fromm, E. (1976): *The Forgotten Language; An Introduction to the Understanding of Dreams, Fairy Tales, and Myths.,* Henry Holt & Co.

Hahn, M. and Goeschl, S. (2004): Blowfishj. Blowfish encryption for java http://blowfishj.sourceforge.net/index.html.

Kaplan, C. S. (2000): Taprats www.cgl.uwaterloo.ca/~csk/washington/taprats/.

Kaplan, C. S. and Salesin, D. H. (2004): Islamic Star Patterns in Absolute Geometry *ACM Transactions on Graphics,* **23,** 97-119.

Schneier, B. (1993): Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). In *Fast Software Encryption*Springer-Verlag, Cambridge Security Workshop Proceedings. 191-204.