Taming the complexity: The need for program understanding in software engineering

Raghvinder S. Sangwan, Ph.D.

Pennsylvania State University, Great Valley School of Graduate Professional Studies

Robert Glass notes that one of the fallacies of software engineering education is teaching people how to program by showing them how to write programs [1]. Corbi also points out that unlike classical language disciplines such as English where students are taught basic language skills and writing techniques, and required to read and critique various authors before they can go on to become copy editors or authors for major publications there is a lack of similar education for computer science and software engineering students [2]. Software engineers learn about software design principles, patterns, paradigms and programming languages, and are expected to produce high quality designs and code, often without ever having seen good examples. Learning by reading is an underused method in computing but is used effectively in many other disciplines.

It has been suggested that it may be so because companies protect the code for their software systems as a trade secret and so, there has been a lack of real-world high quality code to study. The situation is, however, changing with the availability of large number of high quality open source software systems. The first part of this paper describes a course developed by the faculty of Pennsylvania State University which seeks to educate graduate software engineering students in program understanding techniques with the objective that they can now learn to write good quality code by reading and critiquing these open source systems. In addition, these techniques can help them understand and possibly modify

publicly available code. They can learn to use the code as a specification when no other reliable documentation exists.

The need for software engineers to acquire the skill set to understand and critique software systems is becoming increasingly important for other reasons. Many organizations still rely on legacy systems, and more often than not these systems are significantly large and complex. As they become large and complex, it is usually the case that the legacy systems have outdated or little supporting documentation and the engineers who worked on them have long since left. It, therefore, becomes necessary to extract high level design from low level code to better understand these systems and periodically restructure them to meet future needs. If not evolved systematically, these systems can likely become too complicated making future maintenance and evolution activities difficult and cost prohibitive. The latter part of this paper shows the systematic application of the program understanding techniques on an open source software system to demonstrate how these techniques can be effective in managing complexity as a system evolves over its lifetime.

Program understanding in software engineering curriculum

The course in program understanding at the Pennsylvania State University exposes the graduate software engineering students to the techniques and strategies for understanding and analyzing large software systems. Through program slicing, reverse engineering and software visualization they learn to construct abstract representations of the system that can be explored in a systematic way. Through this exploration, they begin to discriminate between systems that are inherently complex and those that are unnecessarily complicated. Such insights are followed by techniques to transform a system to a more desirable form.

Table 1 gives a high level overview of the course modules, their corresponding topics and learning outcomes.

Table 1: Program understanding course model

Module	Topics	Learning outcome
Program understanding problem	Program interleaving	Demonstrate in exposition the complexity of program understanding
	Computational complexity	
Program understanding techniques and strategies	Program slicing	Demonstrate in understanding programs the ability to use different techniques and strategies
	Reverse engineering	
	Software visualization	
Assessing quality of software design	Design principles	Demonstrate in assessing programs the knowledge of good design principles and software quality metrics
	Software metrics	
Improving quality of software design	Refactoring & continuous design	Demonstrate in transforming programs the knowledge of strategies for code restructuring and enhancement
	Testing and migration	

The first module starts with an overview of the program understanding problem. Students are shown that one of the factors making understanding programs difficult is program interleaving [3] – contiguous sections of code can often contain fragments intended to accomplish seemingly unrelated tasks. Program understanding problem is computationally difficult [4] and, therefore, it is a challenge to reconstruct the architecture and recover the design of a system from its low-level code.

Given this background, program understanding techniques and strategies, such as program slicing, reverse engineering and software visualization, are introduced in the next module. Program slicing is a decomposition technique that extracts from a program, statements relevant to a particular computation [5]. Reverse engineering is used to identify system components and their interrelationship (architecture reconstruction) and creating representations of a system at a higher level of abstraction [6]. Software visualization makes an intangible software system that has no physical shape or size visible by using graphical techniques that display programs, program artifacts and program behavior [7].

The next module in the course deals with assessment of the quality of software design. Object-oriented design patterns / heuristics [8] are discussed and metrics [9] are introduced that measure aspects of a system that are demonstrably good. Most positive properties of a system are, however, qualitative and not quantitative making them difficult to measure. These include qualities such as performance, reliability, availability, security, testability and usability. The goal-question-metric paradigm [10] is discussed as a possible mechanism that can be used for systematic specification of metrics under such circumstances.

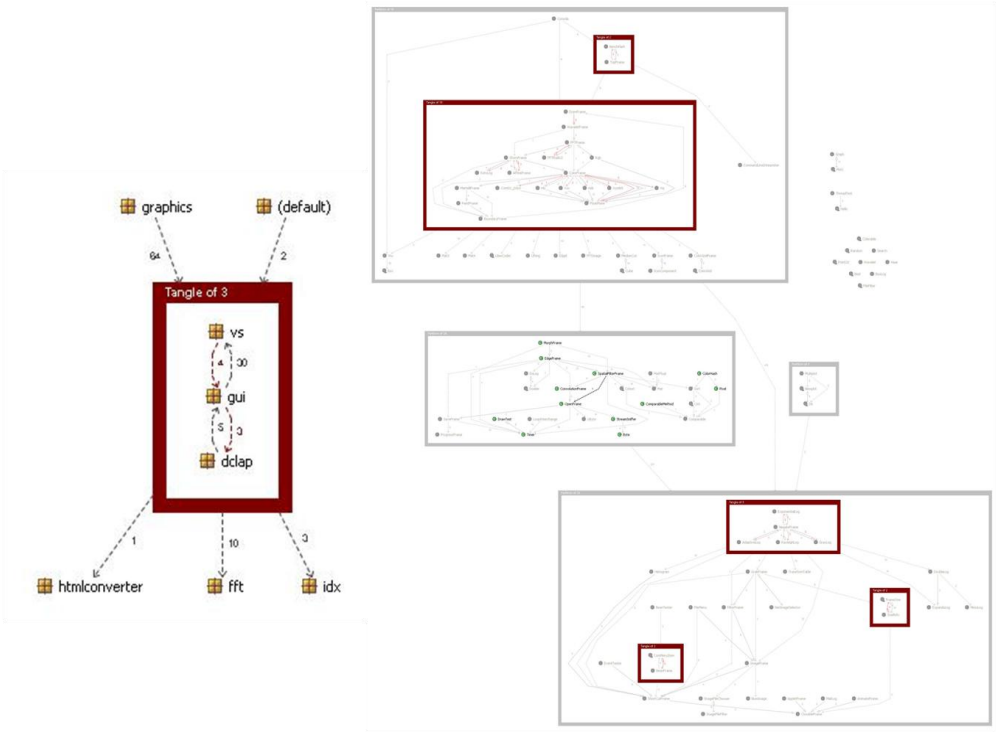
The final module discusses the techniques for improving the quality of software design. Refactoring is introduced as a behavior preserving transformation that improves the internal structure of the system [11]. Testing is also introduced as a strategy to enable software evolution while minimizing the risks of change [11].

As a part of the course, students are also assigned a project where they are asked to select and study an open source software system using a combination of tools for program comprehension, transformation, and design and quality assessment.

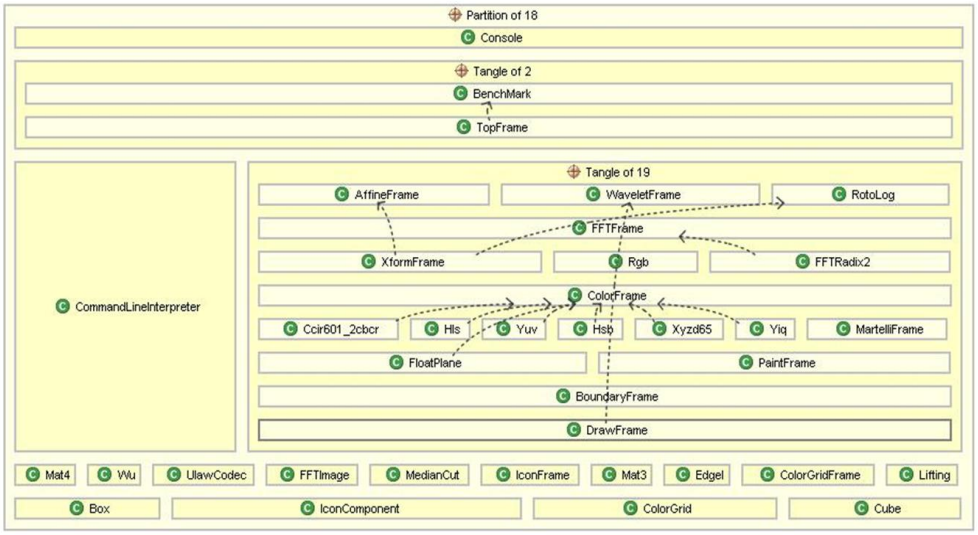
Program understanding in practice

Program understanding techniques provide means for software engineers to systematically investigate and understand systems using their code as the specification when no other reliable documentation exists. These techniques can also help uncover the hidden complexity in the system prompting steps to prevent it from becoming too complicated. We put these ideas into practice by reengineering an existing open source imaging system, Kahindu, to make it more maintainable, reusable and generally simpler to understand and extend [12].

In order to analyze Kahindu, we used a code comprehension and analysis tool called Structure101 (http://www.structure101.com). Structure101 can reverse engineer an existing software system creating a high level abstract model of its structure. At its highest level, the model is represented as a hierarchical directed graph showing system modules and their relationships. One can progressively drill down each module revealing its substructure and at the lowest level, the constituent software classes, their attributes and methods. The model for Kahindu is shown in Figure 1.



(a) (b)



(c)

Figure 1: (a) Module dependency graph for Kahindu. Nodes represent modules and edges represent dependencies; tangles are marked, with highlighted edges representing the

minimum feedback set. (b) Structure of the `gui` module. (c) Conceptual architecture of a partition within the `gui` module

Figure 1(a) shows the model at highest level of abstraction with eight modules. Dependencies among modules are shown as arcs with the labels on the arcs representing number of dependencies. For example, the `gui` module has 30 dependencies on the `vs` module and the `vs` module has 4 dependencies on the `gui` module. The model also shows tangles – groups of modules directly or indirectly dependent on each other due to cyclic dependencies. For example, at this level, Kahindu contains a tangle of 3 consisting of `vs`, `gui` and `dclap` modules. The highlighted edges (edge from `vs` to `gui`, and `gui` to `dclap`) within a tangle represent a minimum feedback set – if the dependencies represented by these edges are removed, the tangle goes away. Cyclic dependencies are not desirable as they make the design of a system rigid, fragile and difficult to reuse.

Drilling down into the modules of Kahindu shown in Figure 1(a) revealed `gui` to be the most complex module – this single package represents 70% of the entire code base and contributes significantly to the excessive complexity in Kahindu. The structure of this module is shown in Figure 1(b). Due to limited real estate, it is hard to see the names of the classes, the number of their dependencies and the tangles (in the tool you can zoom into the area of interest). The structure, however, does give one a sense of a complicated module consisting of over 5000 dependencies and 5 tangles. In addition to tangles (shown in brown), Figure 1(b) also partitions the graph into clusters (shown in gray). A cluster groups nodes that are close together in a dependency graph suggesting a cohesive group of classes that is loosely coupled with classes in other clusters and, therefore, can potentially be separated into its own module. There are 4 clusters in this figure.

Each partition in Figure 1(b) can be examined more closely by laying out its conceptual architecture. We show this in Figure 1(c) for one of the most offending partition in the `gui` module. The conceptual architecture uses an arrangement of cells with the top down structure indicating layering – cells should be used only by cells in the higher layers. Dependencies across cells that break this principle are shown as dotted arrows. There are 27 such violations (each arrow represents one or more violating dependency).

The Kahindu system can be further analyzed in detail using object-oriented metrics such as those shown in Table 2.

Table 2: Metrics for the Kahindu system

Metric	**Measure**	**Analysis**
Response factor for a class	Average: 13.503 Maximum: 103	High response factor makes classes difficult to understand, test and debug
Depth of inheritance hierarchy	Average: 1.567 Maximum:19	Deep inheritance hierarchy implies complex design that is harder to understand and test
Data classes	25	Data classes break encapsulation
Feature envy classes	8	Feature envy classes break encapsulation
Large classes	Average method count: 9.737 Maximum method count: 75	Too many responsibilities packed into a class making it incohesive

Overall then it is evident that Kahindu's design is overly complex and is need of major refactoring. To improve its design, we started with the basic requirements for Kahindu. Kahindu is used for displaying images stored in various file formats such as PPM, JPEG and GIF. These images are then transformed in a number of ways such as making the image lighter or darker, converting color image to grayscale, creating a negative image, filtering or sharpening the details in the image and outlining the boundary of objects in the image via edge detection. This is a sequential process implying there may be a need to undo one or more steps, or starting all over again from the original image if a given sequence of transformations does not produce the desired result. Apart from these basic functional requirements, the system needs to be extensible such that future requirements to handle

additional file formats, additional filters and transformation algorithms can be easily accommodated.

Once the requirements were understood, we created a domain model for Kahindu that showed the real world conceptual classes in the image processing problem domain and their interrelationships. The goal here is to use this model as a motivator for designing software classes reducing the representation gap between how the world of image processing is perceived and how a system designed for this world is implemented. The domain model is shown in Figure 2(a). The fundamental conceptual classes in this model are an `ImageLoader` that loads an `Image` from which an `ImageModel` is created that is subsequently transformed by an `ImageTransform`.

Using the fundamental requirements and the domain model, a design class diagram for Kahindu was created as shown in Figure 2(b).

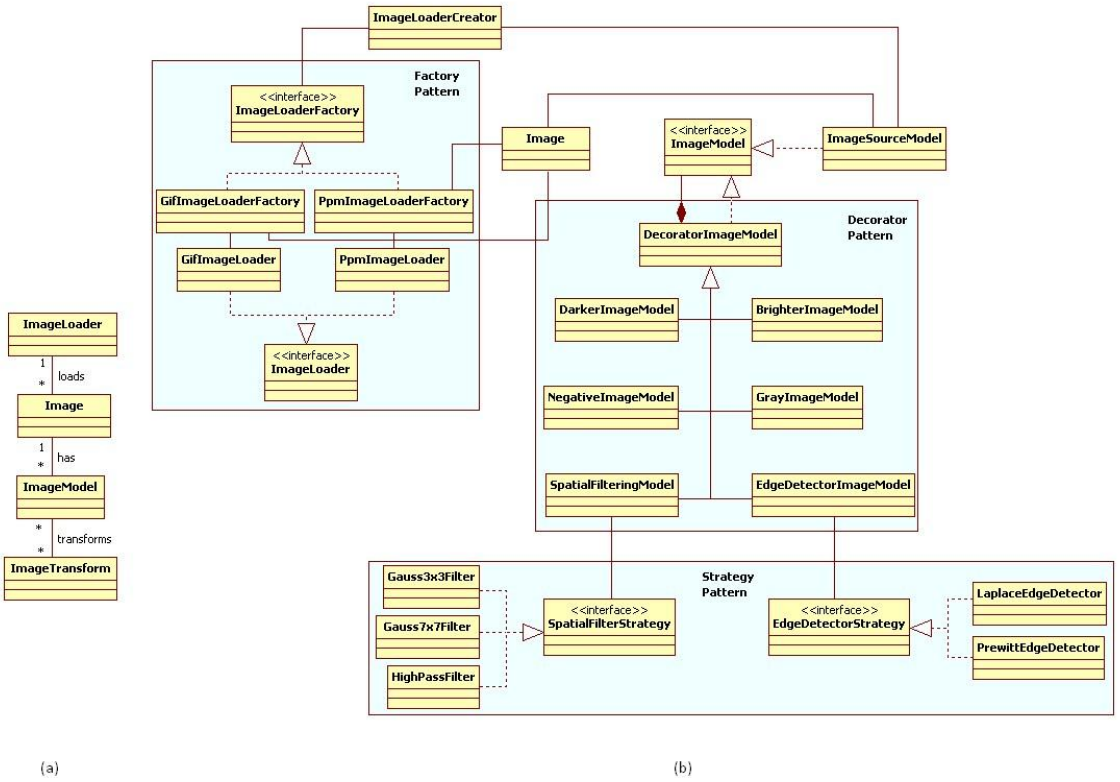


Figure 2: (a) Domain model for Kahindu. (b) Design model for Kahindu.

A number of design patterns were used in the design model to address the non-functional requirement that Kahindu should be flexible enough to easily accommodate future requirements. The patterns include the factory pattern to handle additional file formats, strategy pattern to handle additional filters and transformation algorithms. The decorator pattern was used for chaining together a sequence of transformations on a given image allowing those to be undone if so desired. Once the refactored system was designed, we performed an analysis similar to the original Kahindu system. The high level abstract model for the new system is shown in Figure 3(a). The `loader` module is responsible for loading an image, the `imagemodel` transforms the loaded image, and the `view` module displays the image. The `utils` module contains utility functions used for timing measurements of

the various transformation algorithms. Unlike the original Kahindu system, the new system has no tangles at this level.

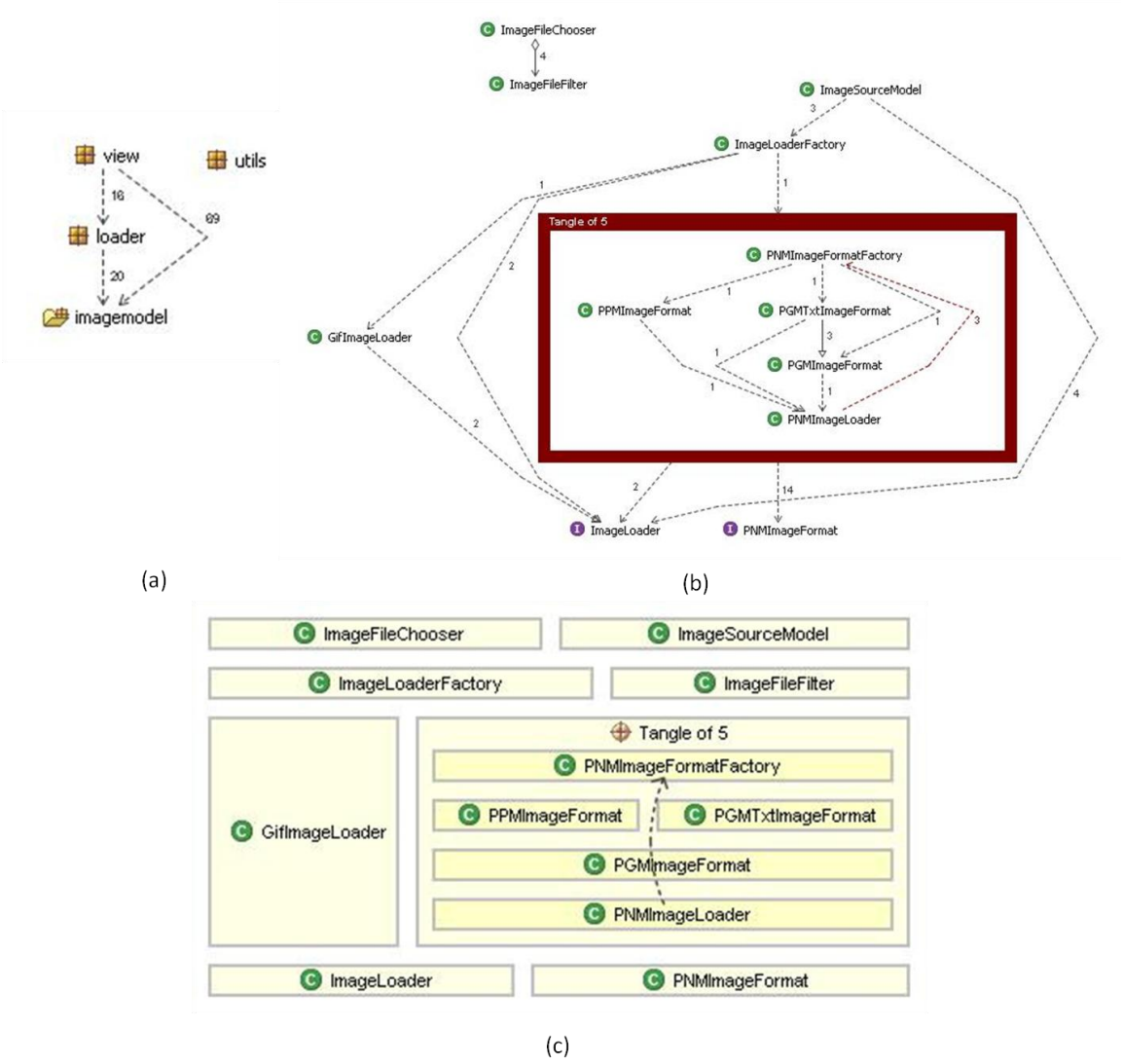


Figure 3: (a) Module dependency graph for refactored Kahindu. (b) Structure of the `loader` module. (c) Conceptual architecture of the `loader` module.

Drilling down into the various modules reveals `loader` module to be the most complex but much simpler than the `gui` module of the original Kahindu system. The structure of this module is shown in Figure 3(b). The dependency graph at the top in the figure shows

one tangle with a minimum feedback set consisting of a single edge from `PNMImageLoader` to `PNMImageFormatFactory`. The same dependency appears as a violation (shown as a dotted arrow) in the conceptual architecture diagram in Figure 3(c). On closer analysis, one discovers that the PNMImageLoader is using the PNMImageFormatFactory to correctly load its image; therefore, this dependency. Alternative design strategies can be explored to remove this dependency and eliminate the tangle.

Compared to the original Kahindu system, the design of the refactored system is more flexible making it simpler to add new functionality, algorithms and image formats while minimizing code changes and maximizing code reuse. The patterns used result in lightweight, loosely coupled and highly cohesive modules improving the maintainability, reliability and integrity of the resulting system. This significant improvement is also reflected in the metrics for the refactored system shown in Table 3.

Table 3: Metrics for the refactored system

Metric	Measure	Comparative Analysis with the Original System
Response factor for a class	Average: 7.31 Maximum: 54	Reduced by half
Depth of inheritance hierarchy	Average: 0.207 Maximum: 1	Average reduced by a factor of 7 and the maximum reduced by a factor of 19
Data classes	1	Data classes virtually eliminated
Feature envy classes	0	Feature envy classes eliminated
Large classes	Average method count: 5.44 Maximum method count: 35	Method count reduced by half

Conclusions

One of the objectives of the program understanding course is to help software designers and developers become more effective in doing design and code reviews, and introduce software

architects to techniques and strategies for architecture reconstruction and for monitoring systems for architectural conformance. The graduate professional students who enroll in this course have found it to be very useful in this regard and student evaluations of the course assessed through official university course evaluation instruments have been high.

Summarizing student comments and feedback several useful outcomes are apparent. They find this course most useful for projects involving reengineering of legacy systems. Techniques in program understanding are useful for software maintenance as well; more than half the time during maintenance is spent understanding the system. Students feel that they are now better armed to not only factor in this time for project deliverables but also educate the project management community on the importance of integrating program understanding tools, techniques and strategies into their software development projects. The students, however, find that no single technique in itself is sufficient but a number of program understanding techniques combined together are more effective.

Program understanding is a hard problem to solve. This makes it challenging and expensive to work with poorly constructed legacy systems. As the proverb goes, "Pay me now or pay me much more later." It is important to stress then that we must focus on creating systems that are easier to understand, maintain and enhance in the future. This is the most significant objective of the course, and techniques for manual code reading, software visualization, and automatic and semi-automatic approaches to assessment and improvement of design and code quality go a long way in supporting this goal.

Acknowledgement

I would like to thank Drs. Phillip Laplante and Colin Neill for their support in developing this course and Dr. Stan Rifkin for providing valuable suggestions in improving this manuscript.

References

1. Glass, R. *Facts and Fallacies of Software Engineering*, Boston, MA: Addison-Wesley, 2002.
2. Corbi, T. Program Understanding: Challenge for the 1990s, *IBM Systems Journal*, 28(2), 1989, pp. 294 – 306, 1989.
3. Rugaber, S., Stirewalt, K., and Wills, L. The Interleaving Problem in Program Understanding, *Second Working Conference on Reverse Engineering*, Toronto, Ontario, Canada, July 14 – 16, 1995, pp. 166 – 175.
4. Woods, S. and Yang, Q. Program Understanding Problem: Analysis and a Heuristic Approach, *IEEE Proceedings of the International Conference on Software Engineering*, Berlin, Germany, March 25 – 29, 1996, pp. 6 – 15.
5. Weiser, M. Program Slicing, *Proceedings of the 5th International Conference on Software Engineering*, San Diego, California, USA, March 1981, pp. 439 – 449.
6. Chikofsky, E. and Cross, J. Reverse Engineering and Design Recovery: A Taxonomy, *IEEE Software*, vol. 7, no. 1, January 1990, pp. 13 – 17.
7. Lanza, M. *Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization*, Ph.D. Thesis, University of Berne, Switzerland, 2003.
8. Riel, A. *Object-Oriented Heuristics*, Reading, MA: Addison-Wesley, 1996.
9. Chidamber, S., and Kemerer, C. A Metrics Suite for Object-Oriented Design, *IEEE Transactions on Software Engineering*, 20(6), June 1994, pp. 476 – 493.
10. Basili, V. and Weiss, D. A Methodology for Collecting Valid Software Engineering Data, *IEEE Transactions on Software Engineering*, vol. 10, 1984, pp. 728 - 738.
11. Demeyer, S., Ducasse, S., and Nierstrasz, O. *Object-Oriented Reengineering Patterns*, San Francisco, CA: Morgan Kaufmann, 2003.

12. Sangwan, R., Ludwig, R., Neill, C. and Laplante, P., "Design Improvements and their Impact on Performance of an Imaging Framework," *Journal of Imaging Science and Technology*, Volume 49, Number 2, March/April 2005, pp. 154 - 162.