Using Context Information to Re-architect a System*

Laura Ponisio, Oscar Nierstrasz
Software Composition Group
University of Bern
Switzerland
{ponisio,oscar}@iam.unibe.ch

Abstract

Successful software systems cope with complexity by organizing classes into packages. However, a particular organization may be neither straightforward nor obvious for a given developer. As a consequence, classes can be misplaced, leading to duplicated code and ripple effects with minor changes effecting multiple packages.

We claim that contextual information is the key to re-architecture a system. Exploiting contextual information, we propose a technique to detect misplaced classes by analyzing how client packages access the classes of a given provider package. We define locality *as a measure of the degree to which classes reused by common clients appear in the same package. We then use locality to guide a simulated annealing algorithm to obtain optimal placements of classes in packages. The result is the identification of classes that are candidates for relocation.*

We apply the technique to three applications and validate the usefulness of our approach via developer interviews.

Keywords: Packages, software measurement, simulated annealing, program understanding, reverse engineering

1 Introduction

Where was *that* class that I always forget to update each time I make a change on *this* one? Where was the class that was doing something similar to what this one does? Not having this information at ones fingertips results in delays, duplication, and bugs which lead to seriously hindering the maintenance of an object-oriented system [11].

Class management, however, can be facilitated by the intelligent use of a group of classes. In the context of this paper, a *package* is a group containing definitions of classes. Packages are important because they are units of organization [8]. Years of modularization folklore encourage developers to put related classes in the same package to make systems more flexible with respect to changing requirements. However, as the system evolves, the organization of classes into packages degrades. Different clients have different views regarding which classes are related, and therefore, in which package a class should be defined. In other words, the *locality* of a *group of related classes* is altered or destroyed by dispersing the classes into different packages. This different perspectives transform the original design into an unclear one, which in turn hinders the maintenance of the system. The question now is how to re-gain a high locality of the classes and how to understand the locality and the whole system [1].

Previous approaches neglected the perspective of the client. We claim that context information is the key issue to re-architecture a system. To base understanding only on internal attributes of the software entities is not enough. We have also to look at the way the software entities are *used*.

We have developed a technique to detect classes that are misplaced. Part of the challenge was to answer the question: misplaced with respect to *what*? If different viewers of a set of classes have different interests, it is not realistic to suppose that developers would agree about what is the ideal way of partitioning the system. And if we cannot measure it yet, how can we reason about it?

Contextual information indicates similarities between classes through the way classes defined in a package are used. Based on this information, our technique uses a visualization that points out classes potentially misplaced.

The user of this technique benefits by getting a better understanding of the system. In the first place, getting a mental picture of the overall state of the system regarding locality of classes. In the second place, the developer benefits by getting hints of which classes are misplaced, where they potentially belong, and which misplacement to tackle first. Our approach does not automatically apply changes to the code.

We validate this technique by applying it to three real

*In Proceedings of the 3rd Software Measurement European Forum 2006 (SMEF'06), 2006, pp. 91-103.

systems, describing how they communicate design principles violation and how they help the developer to understand the system in common software analysis tasks.

Structure of the paper. In Section 2 we present our approach in a nutshell. In Section 3 we explain how we represent packages. In Section 4 we define locality. In Section 5 we present the optimization algorithm used to obtain solutions with maximum or near maximum locality. In Section 6 we show the results of applying out methodology to the case studies. In Section 7 we elaborate some discussion and future work. In Section 8 we refer to related work before concluding in Section 9.

2 Context Information in a Nutshell

Our ultimate goal is to improve the system in the sense that classes that are used together are in the same package. As a result, we propose an operational technique (a procedure) to represent and manage the location of classes into packages in large object-oriented systems. But first we have to understand and represent how classes in a package belong together. We exploit the information present in the context, *i.e.,* how the classes are used. Let us present a metaphor from every day life that describes the idea behind our approach to catch the locality of the classes.

Example Books related to the subject $green$ may belong to different editorial offices, have different authors and have, in short, no explicit attribute that connects them, but they also belong to a group: the group of books *consulted* by readers of subject $green$.

If we were organizing a library, it would make sense to put on the same shelf the books that belong to the same field of study. That is straightforward if we know the field of each book, but not if we are unaware of it. However, even without knowledge about the contents of the books, we could observe that every group of readers is interested mainly in one field. If that is the case, a book read only by people interested in subject $blue$ that is on the same shelf as books read by students interested in subject $green$ would call our attention and most people would agree that it is out of place. Figure 1 depicts this situation.

We expand this idea to understand locality in large object oriented systems. More concretely, to describe the locality of the classes of a package.

Now Figure 1 does not represent books anymore, but code. It depicts four packages of a system containing classes. We believe that most of the developers would agree that the locality of these packages would increase if the *blue class* where defined in the *blue package*, instead of being defined in the *green* package.

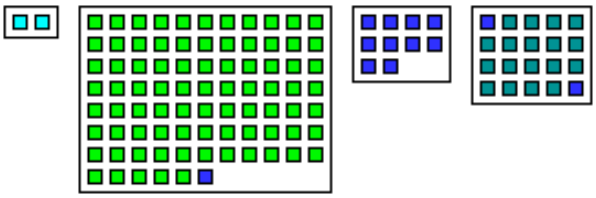

Figure 1. In this library, among the green books there is one read by the people that read only blue books.

This example presented the idea behind our technique to find classes that are misplaced: two classes are related if they are used by the same package. We define locality of a package as a quantification of how much its classes are related.

There are conditions under which this quantification holds. Indeed, there are some restrictions that determine if a package is a good indicator. For instance, if a package P accesses every class in every package, then P does not provide information. Our solution in this case is simply to ignore P.

3 Package Representation and Analysis

Software doesn't have any form. Opposed to mechanical simulation, where real objects are represented in two or three dimensions, software is intangible [10]. We choose a *rectangle* as the shape to represent packages and classes [7].

The goal is to capture characteristics of the packages or classes that we consider relevant for understanding the code, and to associate them through metrics with the rectangle size and color. Figure 2 shows the mapping from code to our visualization.

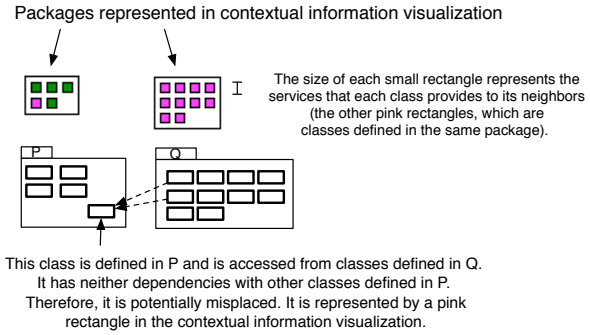

Figure 2. Two packages and their representation in contextual information visualization.

The big rectangle represents a package, and the small ones inside it represent the classes *defined* in the package. The color shows the locality of the classes. If the class is misplaced, it is drawn as a rectangle inside the package where it is defined and painted with the color of another package (a potential destination).

3.1 Contextual Information Inside the Package

Understanding the distribution of classes into packages is one way of analyzing the pros and cons of re-structuring a large-scale software system.

With visualization described in the previous section, the developer receives hints about which classes to move and where, but nothing about what he should tackle first.

Contextual information provides a solution to this problem. It captures this pulling in forces with the *happiness* of a class.

3.2 The Happiness of a Class

Classes interact to perform tasks. They are naturally sociable. Therefore they are *happy* when they have clients in package where they are defined. The more services a class provides to its neighbors, the happier it is.

With the locality, we detect classes whose clients belong to foreign packages. When a class is used by some foreign clients and the neighbors are used by a complete different group, this circumstance pulls the class *out*. But this information omits the reasons why the class is in the package. For instance, there can be structural reasons supporting the placement of a class in a determined package.

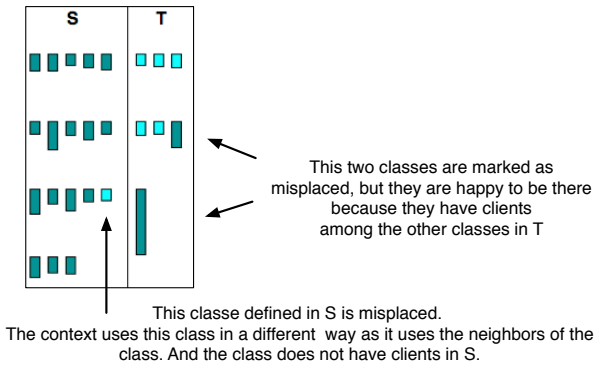

Figure 3. Visualization of the happiness of a class given by the context information.

By adding size to the rectangles representing classes, the proposed visualization can show not only the misplaced classes, but also hints that tell the developer which movements are more important than others.

The next section details the procedure to automatically find the misplaced classes and the advice as to where they could be defined.

4 Capturing Locality

Locality captures the degree to which classes in a package P belong together.

Cohesion measures also the degree to which classes belong together [4], the difference being that cohesion is derived through the explicit dependencies between the classes *in* P, whereas locality is derived through the contextual information. More concretely, locality is derived from the way classes are used. It represents the forces of the context pulling classes together or apart.

Given a set $\mathcal{D}$ of explicit dependencies between classes defined in packages, *locality* counts how many classes in the package are used by the same client. In Figure 4(b) all the classes are accessed by classes in the two client packages. But in Figure 4(a) class H seems to not to belong to the package $P1$.

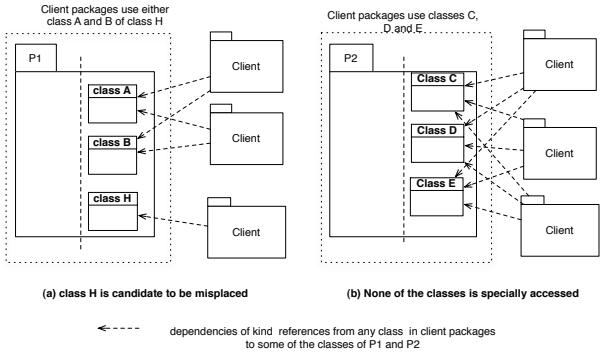

Figure 4. Deriving locality from the context.

4.1 The Model

Our source model consists of classes, packages, and dependencies. To express the cohesion measures unambiguously we provide the following set-theoretic formalism.

An object-oriented system consists of a set of classes, $\mathcal{C}$, where A, B, C range over classes.

$$A, B, C \in \mathcal{C}$$

Let $\mathcal{P}$ be some partitioning of $\mathcal{C}$, where P, Q, R range over partitions.

$$P, Q, R \in \mathcal{P}$$

There are dependencies between classes. Each dependency is of kind references or inherits.

$$\mathsf{inherits}, \mathsf{references} \subseteq \mathcal{C} \times \mathcal{C}$$

Each dependency determines a client and provider

$$\mathsf{depends} \subseteq \mathcal{C} \times \mathcal{C}$$

The *clients* of a class are the classes that depend on it:

$$\mathsf{clients}(C) = \{A \in \mathcal{C} \mid A \ \mathsf{depends}\ C\}$$

A partition P may contain classes that have clients in other partitions. These classes constitute the interface of P.

$$\mathsf{interface}(P) = \{C \in P \mid \mathsf{clients}(C) - P \neq \emptyset\}$$

The classes of P that do not belong to the interface of P are internals and we ignore them.

inherits represents the single-inheritance definition, the subclass being the client being and the superclass being the provider.

references, represent explicit references to a class such as the ones created during class instantiation.

Finally, we acknowledge every dependency between two classes. For instance, if A instantiates B three times, then we have three dependencies between A and B where A is client of B.

4.1.1 The Definition of Locality

We need a way to quantify the special accesses on a class regarding those of its neighbours. This is the definition of locality of a package.

Definition 1 *We define Locality (*loc*) of a package as the sum of pairs of classes from the interface of a package having a common client package (*f*), divided by the number of pairs that can be formed with all classes in the interface.*

$$\mathsf{loc} = \sum_{a,b \in I} \frac{\mathsf{f}(a,b)}{\#Pairs}$$

Where

$$\begin{array}{lcl} I & = & \mathsf{interface}(P) \\ \#Pairs & = & \frac{|I| \times (|I|-1)}{2} \\ C & = & \mathsf{clients}(a) \cap \mathsf{clients}(b) \\ \mathsf{f}(a,b) & = & \begin{cases} 1, & \text{if } C \neq \emptyset \\ 0, & \text{otherwise} \end{cases} \end{array}$$

Note that if $\#Pairs = 0$ (*i.e.,*, if $\mid I \mid = 0$ or 1), then loc is undefined, since we cannot infer anything from the context if the context does not use the package.

But given a new configuration of classes in the system, we want to know how is the locality for the whole system, to know if we are doing better or worse by moving the classes. We quantify that with the average of loc applied to the packages.

Definition 2 *We define average locality of a system* avloc *as the average of the values of* loc *of the packages.*

$$\mathsf{avloc} = \sum_{p \in \mathcal{P}} \frac{\mathsf{loc}\ (p)}{\mid \mathcal{P} \mid}$$

We determine manually which are the packages that should participate in the analysis. Packages without clients are excluded because we cannot infer the locality of its interface classes. Packages that access ubiquitously classes in other packages are ignored because their dependencies do not provide meaningful information. For example package P that access only three packages, tells about the locality of the classes defined in those packages. However, P accessing every class in every package in the system carries poor information about those accessed packages.

5 Simulated Annealing Guided by Context Information

In 1983 Kirkpatrick et al. [9] proved that the search of an optimal solution in combinatorial optimization algorithms is analogous to a method that models the search of a state of balance of a solid. In the field of thermodynamics, that method is well know as *simulated annealing* (SA).

When using simulated annealing to make glass, for example, the system starts at a high temperature with molecules moving randomly. The higher the temperature, the more the molecules move. After some time the system cools down. Eventually, it cools down enough for the molecules to form a glass, which is the resulting solid. If the glass contains defects, the system is heated again followed by the cooling time. This process is repeated until finding a satisfactory result, in our example, a glass with none or with acceptable defects, or having reached a condition to stop.

Because of the analogy existing between the method that models the search of a state of balance in a solid, and the combinatorial optimization method, the later was named simulated annealing as well.

We use the simulated annealing method to find an optimized solution to the problem of locality. First, we define the problem as distributing classes into packages maximizing locality in large systems in reasonable time. Then we provide a function to guide the algorithm towards the optimal solution and the possible movements of classes between packages.

The simulated annealing algorithm starts then to search for the optimal solution, or for solutions close to the optimal one.

However, our interest is not in finding the optimal solution. Even if this solution could be found, nothing guarantees that all the developers will agree that this is the ideal partitioning of the system.

Our interest is in obtaining the movement of classes around packages that led the optimization function to be closer to the optimal value. It is this information that we visualize.

The movements In the context of thermodynamics, molecules move randomly. In the context of using context information to re-architecture a system in software engineering, the movement of molecules represents the movement of classes from one package to another. In a software system there are constraints that determine which classes can moved. Due to this constraints and for reasons of performance, the movements in our approach are guided rather than random.

Experiments giving total freedom in choosing randomly the next package to be acted upon were inefficient. We therefore optimized the algorithm. The modification consisted in ordering the packages according to a criterion, for instance the cohesion of the package, and apply random movements on those packages with lower cohesion.

The Optimization Function The simulating annealing algorithm needs an objective function to guide it to find solutions tending to the optimal one. Because our focus is to understand the locality, we define the optimal solution as the one that maximizes the average locality of the system. The optimization function is in this case (1 - avloc), which is based on avloc as it was defined in Section 4.1.1. The objective function ranges between 0 and 1.

6 Analysis of the Context Information

In order to study the effectiveness of our approach, we applied in three real systems to perform common tasks in reverse engineering such as the analysis of coupling and cohesion, and the understanding of the architecture.

Code should always exhibit low coupling and high cohesion, and the architecture of the system (*e.g.,* layered, blackboard, etc.) should be respected. In the following examples we apply the proposed contextual information approach to real systems and show how we use it to perform the analysis tasks.

6.1 SYSTEM I

SYSTEM I consists of 138 classes distributed into six packages. It has a layered architecture with package Q and S being in the lowest layer, and packages P and R in the layer immediately above. All its subsystems use a common structure defined in package S. Figure 5 shows the locality of SYSTEM I obtained using the contextual information.

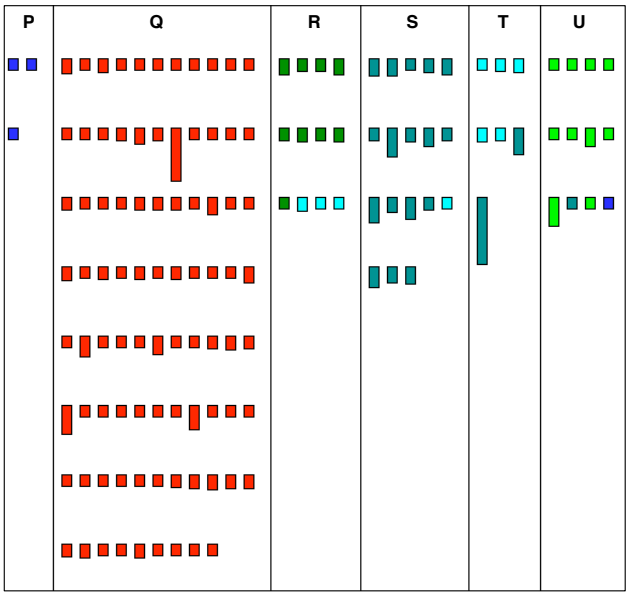

Figure 5. SYSTEM I Example. More than one color in a package indicates potential lack of locality. Size of smaller rectangles represent forces pulling the class towards the package.

6.1.1 Pattern 1: The Library

We observe a significantly bigger package, Q containing 86 classes, more than the half of the classes of the system. Most of the classes in this package are not connected to each other. We see only one rectangle significantly bigger than the others. This represents an exceptional class who is related to the others by being their superclass or referenced statically (*e.g.,* instantiated).

The small size in most of the red rectangles show that coupling between the classes in package Q is low. However, the visualization shows that there is no evidence that moving its classes somewhere else would increase the average locality of the system. If used at all, the shape of Q suggests that it could be, for instance, a big library.

6.1.2 Pattern 2: The Newcomer

If we focus our attention on packages S and T, we observe that there seems to be a close interaction among those. T

depends on S, and S depends on T. The small cyan rectangles in R and S, indicate that T consumes services from classes in package R as well, being almost the exclusive client of classes defined in R and S.

It is surprising that two classes in T call attention towards the locality, but at the same time they have higher coupling with the other classes defined in T. They are, therefore, happy to be defined in T, and their size is a hint to the developer of the presence of a possible anti-pattern or bad smell.

Closer inspection of the code determined that a later addition of methods in package S added a violation of the layered architecture. This methods, belonging to classes in S (which is in the lowest layer), reference explicitly classes defined in a higher-layer package as T. If we know that S is in a lower layer that T, then the picture denounces the existence of dependencies from a package in a lower layer *towards* a package in a higher layer.

This situation was a consequence of T being a later add-on, and those foreign cyan classes in R and S being added only to satisfy needs of the new client, T.

6.1.3 Pattern 3: The Remainer

In package U, most of the classes have low coupling among them, indicating a potentially low-cohesive package. Besides, two of the lowest-coupled classes are hinted to be put in two other different packages. This looks like plainly misplaced classes.

Inspection of the code indicated that they were obsolete classes, one implementing printing debugging facilities to debug the code of a class defined in P, and the other an obsolete builder statically referenced by the class responsible for the user interface. This last class was defined in R. The classes were misplaced, for the developers agreed that they belong to the trash.

6.2 CODECRAWLER

CODECRAWLER is a small software visualization tool [7][1] With 244 classes distributed in eleven packages, CODECRAWLER was built under the same design principles than SYSTEM I, but with the difference that while SYSTEM I is heavily evolving, CODECRAWLER is already a mature tool. Figure 6 shows a visualization of CODECRAWLER. The uniformity of colors show a high modularization of the packages. Only three of the eleven packages show more than their own color, and in all the cases, the classes hinted to be relocated do not have particularly strong coupling with the others in the package, which is a sign of good design.

[1] See http://www.iam.unibe.ch/ scg/Research/CodeCrawler/ for more information.

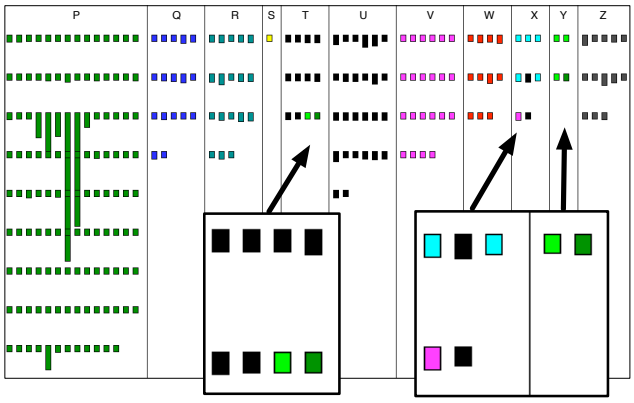

Figure 6. CODECRAWLER Example. Uniformity of one color in most of the packages indicates high modularization. Only in three specific places there are potentially misplaced classes.

6.3 An Industrial System

BASE VISUALWORKS is an industrial system, developed over the last 16 years. It contains the runtime entities of the Cincom VisualWorks Smalltalk environment[2]. These range from the classes, and memory objects to the compiler framework, debugger, code browser, and the support for the operating system. Its 2022 classes are distributed into fourteen subsystems, each containing packages. Of those packages, we chose to analyze `Collections`, which contains 228 classes distributed among eight packages. `Collections` is strategic because it contains the classes related to or supporting the collection structure.

6.3.1 Pattern 4: The Friendly Package

The contextual information visualization in Figure 7 shows high locality in packages P, R and S, and low locality in packages Q, T, U, and W. Half of the packages in `Collections` has high locality and the other has poor or extremely low locality.

The structure of package U is particular: it is an extremely friendly package. The many colors of the rectangles in of U indicate that there are strong forces pulling its classes apart. This means that its classes are *separately* accessed by a significant number of packages of the subsystem, namely four out of the eight .

Three classes that seem to be pulled towards U (the little black rectangles) are defined in packages T and W. However, two of those classes are happy in T. Their size, being

[2] http://www.cincomsmalltalk.com

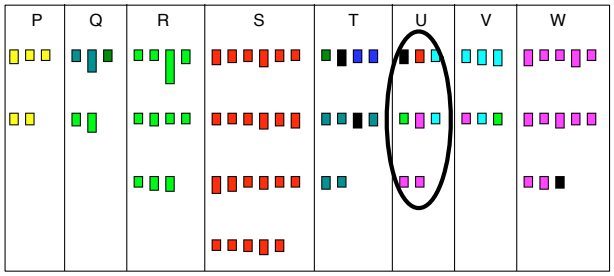

Figure 7. `Collections` **Example. Package** U **shows extremely low locality. The distribution of its classes indicates it is completely anomalous, or a package with a** *special purpose*.

larger than the rest of the classes in T, indicates that they provide services to classes defined in T.

Observed with traditional approaches based only on internal attributes, the classes in U would expose low coupling among them, and U would indicate common low cohesion. Context information shows how exceptional U is. Its classes are so much distributed, that the visualization indicates and anomalous package or a package having a special purpose. It could indicate the existence of a *design decision* supporting the presence of U.

With contextual information visualization showing this pattern, the developer is encouraged to do more investigation before doing a possible dangerous refactoring.

Analysis of code confirmed the uniqueness of U: it contains abstract classes which are root of inheritance hierarchies contained in the neighbor packages, and must therefore be loaded first.

7 Discussion and Future Work

Our approach builds on existing measurements, a simulated annealing algorithm, and a visualization technique.

The novelty of this work's contribution resides in using contextual information of the package, exploiting external attributes as well as the internal ones.

Part of the contribution of this work is the effort of understanding the locality of the system, which resulted in a definition of locality, as a first step to quantify it.

We assume that there is no universal partitioning of the classes in the system that satisfies the different perspectives of all the developers and users of the system. Therefore, we do not attempt to claim in this work that locality is a full measurement. It remains to be proved that locality is an homomorphism from the system entities into numbers, and that it thus satisfy the representation condition of measurement [4]. However, there is intelligence about *some* classes that must be located in the same package, reasoning under the principle that classes that are used together should be defined together.

A Substitute for the Optimization Function The optimization function can be replaced for another based in a established measurement to suit the goals of optimization. For instance, the optimization function can be based on cohesion and coupling metrics, and use different relationships between classes such as inheritance definition. For reasons of space, this task is left as future work.

Scalability This approach has shown to be scalable in two aspects, namely the ability to communicate the intangible picture of the package modularization of large object-oriented software systems, and the flexibility to suit a wide spectrum of goals to optimize, which can be accomplished by substituting the optimization function.

8 Related Work

Visualizing large-scale software to understand its properties is an old and challenging task. Previous software visualization techniques represented entities of code such as classes or packages with a variety of devices, ranging from rectangles [6] to bugs and compass-type plots [**?**, **?**]. This techniques were not originally devised to understand locality, but other aspects of the code. They are too coarse to capture the locality attribute of packages, and they do not exploit the information hidden in the context. Without representing the contextual information, structural properties influencing the distribution of classes remain unseen.

To cope with the problem of poorly understood factors that influence software evolution and reliability, Langelier et al. [**?**] propose to combine automatic analysis with human expertise. This hybrid technique facilitates the analysis, understanding and evaluation of software quality in large-scale software systems. The framework is based on visualization so that human expertise can compensate the limitations of automatic analysis when they are applied to complex software attributes.

Our technique can be used to support the application of the existing scientific methods to the problem of software quality improvement.

A case in point is Basili's Quality Improvement Paradigm (QIP) method [**?**]. Our approach could be used within this method as a concrete technique to understand and improve the goal of optimizing locality in packages.

Of the various levels at which the process of software development can be viewed, if the topmost level is where the engineer thinks about the technology, and the lowest level is where the developer thinks about the implementation details, then the QIP is at a higher level than the technique proposed in this work. However, our contextual information technique can be used to support the learning and feedback activities of software development. It *complements* therefore QIP by providing an automatic method for understanding goals, obtained previously with the application of QIP.

9 Conclusion

The contribution of this article is a novel technique to analyze the locality of classes defined in packages in large object-oriented systems.

We claim that contextual information is key to understand and re-architecture a system.

Traditional approaches ignore the value of contextual information to convey insights about the organization of the classes into packages. However, this work shows how our technique detects misplaced classes from the way the classes are used.

To support the analysis of locality of classes into packages, we define a process and a visualization layout.

Finally, we have applied our technique to three real systems and showed how it facilitates their understanding.

References

[1] V. Basili. Evolving and packaging reading technologies. *Journal Systems and Software*, 38(1):3–12, 1997.

[2] J. Bieman and L.M.Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–658, Aug. 1994.

[3] L. C. Briand, S. Morasca, and V. Basili. Property-based software engineering measurement. *Transactions on Software Engineering*, 22(1):68–86, 1996.

[4] N. Fenton, S. L. Pfleeger, and R. L. Glass. Science and Substance: A Challenge to Software Engineers. *IEEE Software*, (7):86–95, July 1994.

[5] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[6] M. Lanza. Codecrawler — lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*, pages 409–418. IEEE Press, 2003.

[7] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, Sept. 2003.

[8] R. C. Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.

[9] K. S., G. C. D. Jr., and V. M. P. Optimization by simulated annealing. In *Readings in computer vision: issues, problems, principles, and paradigms*, pages 606–615, 1987.

[10] J. T. Stasko and E. Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *INFOVIS*, pages 57–, 2000.

[11] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "RECAST: Evolution of Object Oriented Applications" (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006)