

Coping With Software Change Using Information Transparency*

William G. Griswold

University of California, San Diego
Department of Computer Science and Engineering
La Jolla, CA 92093-0114 USA
+1 (619) 534-6898
wgg@cs.ucsd.edu

ABSTRACT

Designers are frequently unsuccessful in designing for change using traditional modularity techniques. A modularity technique called *information transparency* can improve a designer's ability to plan for change by revealing the interdependence of widely dispersed program elements that must be changed together to perform a change correctly. Unlike information hiding, which represents modules with locality and abstraction, information transparency represents modules by *similarity* and *architecture*. With these, a programmer can create locality with a software tool, easing change in much the same way as traditional modularity. Information transparency techniques include naming conventions, formatting style, and ordering of code in a file. Transparency can be increased by better matching tool capabilities and programming style.

KEYWORDS

Modularity, software architecture, maintenance, software tools

1 INTRODUCTION

Software engineers are frequently unsuccessful in designing for change using modularity and other change localization techniques [Par96]. It is not difficult to anticipate that technology will advance, new industry standards will arise, and competitors will introduce features that will draw customers away. However, appropriately designing for these changes can require understanding the details of these changes, which cannot be known to the designers. Moreover, market pressures dictate that effort be spent in getting the current product to market to generate cash flow and compete with other products, not in reducing the cost of future versions, resulting in a "design for release" mentality. Over time such a scenario leads to rapid escalation of software enhancement costs [BL76].

1.1 Experience with Program Restructuring

Software restructuring (and more broadly, software reengineering) is one proposed solution to this problem [Opd92, GN93]. Software restructuring involves analyzing the structural problems in the software relative to the current and near-future change requirements and then making stylistic,

meaning-preserving, structural changes to the software to evolve its architecture and pattern-level design of the software so that future changes can be performed locally. Our early restructuring research focused on the benefits of meaning-preserving program restructuring, what transformations are needed, and how they can be automated. Later work focused on exploiting the latent structure in programs to guide restructuring changes [BG94, BG98] and other kinds of assistance that do not require expensive and complicated data flow analysis [GCBM96].

Recently we have performed a number of qualitative experiments in restructuring that involved varying levels and kinds of automation [GCBM96, BG97, Gra97, GHK97]. These studies revealed that meaning-preserving restructuring transformations may not be as important as believed. For instance, we discovered:

- Choosing an appropriate redesign for the software is more difficult than performing the global restructuring with standard Unix tools.
- Programmers using Unix tools adopted stylistic methods to making restructuring changes in order to simplify the task and decrease the chance of making mistakes. The change styles are similar to those automated by the transformational tools and offer similar benefits.
- Programmers using Unix tools used time-saving shortcuts if they knew enough about the software, since they were not syntactically constrained by the editor. They sometimes weakened the meaning-preserving criterion, depending on the behavioral constraints of the code being restructured. This simplified the change and sometimes "improved" the software's behavior.
- Although "low-cost, low-tech" restructuring introduced bugs into the software, they were easy to detect and correct, due to the stylized nature of the edits and intention that the changes be meaning-preserving.
- When performing global changes that could not be simplified by mimicking transformational techniques, programmers invented change techniques exploiting the system's architecture and naming conventions, simplifying the change and increasing the chances of its completeness and consistency.

*This research is supported in part by NSF grant CCR-9508745 and California MICRO proposal 97-061 with Raytheon Systems Company.

1.2 The Information Transparency Design Principle

At the heart of these somewhat surprising successes is a not entirely new software design principle that we call *information transparency*, a complement to information hiding. At the module level, it is the property of a module making *visible* its dependence on design decisions that are *not hidden* by a module. If all modules achieve information transparency in a consistent way for a particular design decision they share, then we say the design exhibits information transparency. The consequence is that a programmer, using software tools, is able to easily recognize the connection between widely dispersed program elements that require that they be changed together in order to correct the design or otherwise globally modify the software while not damaging its intended behavior. Information transparency can be achieved through a variety of techniques including naming conventions, formatting style, file naming, and ordering of code in a file. Transparency can be increased by using tools with greater recognition capabilities or by better matching the techniques used to the tools available. If only lexical tools are available, for example, interdependence must be lexically visible.

Information transparency is applicable in at least a couple of ways. Ideally, the code related to a changing design decision is found by identifying the module (e.g., file) that hides the design decision in question. When modularity fails, however, the code must be found by other means, often by a whole-program search using a lexical tool such as Unix `grep`.¹ In effect, using `grep` “brings together” the dispersed code so that the programmer can plan the change “locally”, that is, by being able to look at all the matched code at once, rather than one line at a time in isolation as a text editor would afford. The ultimate change, too, can be achieved in similar ways using an editor and its search-and-replace features. A restructuring may be desired to hide the exposed design decision, but a programmer may choose to change the design decision directly without restructuring.

Another example arises in retargeting a classically designed compiler or programming environment from operating on one language to another, albeit similar, language. We faced this problem in retargeting our star diagram planning tool [GCBM96] from C to Ada [Gra97]. We anticipated restructuring the tool’s software to hide design decisions relating to differences between Ada and C. Although this occurred to some extent in the abstract syntax tree (AST) classes, we found that it was best to apply most changes globally rather than restructure.

For instance, in retargeting the C `while` statement to the Ada `loop while` statement, changes were required in all the major components. Yet the changes—using information transparency—were almost trivial. After downloading an

Ada lexical analyzer and parser from the internet—giving us two major components essentially for free—it was possible to walk through the existing code for C using the architecture as leverage for making the global change: starting in the C parser, references to the `WHILE` terminal were found and copied over to the Ada parser’s use of the `while` and then modified to reflect Ada’s syntax and semantics. Then the programmer moved “downstream” in the architecture to the AST, making any necessary changes to the AST’s `while` statement code, as dictated by the needed actions in the parser and the special characteristics of Ada. Next the programmer moved downstream to the pretty printer, accommodating the changes to the AST code as well as making changes to reflect the differences in Ada syntax. Having made a complete pass over the architecture, the programmer had high confidence that all relevant code had been changed. Moreover, by going through the changes “in order”, the programmer ensured that all design information needed in changing a component had already been gathered. Such a process was simplified by the “pipeline” nature of the architecture, as well as by the fact that the programmer of the C tool had consistently named variables and other constructs “`while`”, `WHILE`, `whileStmt`, etc., when working with the `while` C language construct.

Each such change, although global, was small and independently testable because it was approached in a syntax-directed fashion. This approach increased the number of visits to files, as each pass required visiting perhaps a half-dozen files, but the advantages of atomicity, completeness, and testability fully compensate.

As this example attests, it is difficult or even undesirable to hide all design decisions in small modules. Combined with information hiding, information transparency promises to give programmers much better coverage of the design decisions of interest. Moreover, when a module fails to hide a design decision, it is possible to opportunistically fall back on information transparency to ease the change—if the programmer can discover the underlying patterns that makes the design decision visible.

In the following sections we present some basic principles of information transparency and discuss the implications of information transparency on the design of software analysis tools. We argue that the transparency of a design is increased by programmers and software tool designers catering to the others’ capabilities.

2 INFORMATION TRANSPARENCY PRINCIPLES

As argued in the introduction, some design decisions are either difficult to hide or are best not hidden in information-hiding modules. A simple example is the exported interface of a module itself. Although a module interface can be hidden with another module, the fact remains that this second module’s interface is then not hidden. Although module interfaces might be intended to be stable, the fact is that

¹The Unix program `grep` is a software tool that performs character-based regular-expression matching for each line in one or more files [Aho80]. It has options for tagging output lines with their file and line number, printing only the file name, etc.

they often do (or should) change in response to evolutionary changes to the software. Another example of a commonly exposed design decision is the protocol that the uses of a module must adhere to, the global use of built-in programming language data types, or a globally enforced architectural rule about component interfaces such as conformity to a pipe-and-filter paradigm.

Information transparent design is related to module guides [PCW84]. To cope with the problem that not every design decision can be hidden in a small module, Parnas introduced the concept of designing with nested modules documented with a tree-structured module guide. By allowing some “big” but relatively stable design decisions to encompass several subordinate, smaller, and less stable design decisions, it is possible to limit the cost of making changes to a complex system. The module guide lowers the cost of changes by directing programmers to the submodules that are affected by a particular design change. Information transparency effectively encodes the module guide into the software by using similarity and architecture, thereby allowing tools to aid in identification of interdependent code and also permitting non-hierarchical relationships amongst modules.

The desire to achieve fast and accurate identification of the code related to a global change leads to the three primary principles of information transparency:

- *Program elements likely to be changed together during a complete and consistent change should look similar.*
- *Program elements unlikely to be changed together should look different.*
- *Similar code must be identifiable using available software tools.*

The effect is that widely dispersed program elements can be easily and quickly brought together into a single view, in effect achieving a localized module-like view of the related elements. Although the modification itself is dispersed, automation here as well can provide the benefits of locality. Although the changes may not be isolated to the initially identified code, the cascading effects can also be handled with information transparency. The vital role of tools is considered in the next section.

Simultaneously achieving similarity and differentiation is non-trivial and sometimes counterintuitive. Programmers frequently reuse variable names or add meaningless prefixes or suffixes to distinguish identifiers. In object-oriented systems it is common to reuse method names across classes even when the classes are not related by type. In the introduction, the designers of the star diagram planning tool were careful (or lucky) to choose names across components that reflected the details of the concrete syntax associated with the conceptual object being manipulated. For purposes of language neutrality, the conceptual interface of the abstract syntax tree (AST) could have been much more generic (e.g., us-

ing `loop` or `control_statement` rather than `while`). Another example of similarity is the consistent formatting of interacting groups of statements—for example, a group of statements satisfying a messaging protocol.

Techniques for self-documented code, in particular naming and formatting conventions, are already in wide use in industry, but these are usually adopted for reasons other than information transparency, and so may not improve a design’s transparency as much as desired. Most are intended to ease the readability of code by humans, not tools.

The Hungarian variable naming notation is a convention that was designed to inform or remind a programmer of how a variable is to be used, especially when the type system is weak. For example, a variable name (e.g., `ASTpVHSet`) encodes the structure of the object (i.e., a set implemented as an open hash table of pointers to AST class objects). If a global change revolves around these particulars, then the naming convention can be of use in information transparency. More advanced uses of Hungarian notation include an additional descriptive part that the programmer can use to denote the variable’s role (e.g., `deletedASTpVHSet`). Hungarian notation can be adapted to the needs of information transparency by adding parts to an identifier to denote the relationships (i.e., design decisions) that it shares with other variables, types, etc., in the system. Consequently, a tool like `grep` can find all identifiers necessarily involved in a global change by searching for the tag that denotes the design decision of interest, achieving the goal of similarity. Likewise, identifiers not related to a particular change should customarily *not* include that tag in their names, thereby achieving differentiation.

A peculiar example of information transparent design is copy-paste programming, or what has been called *reuse of uses* [RC93]. In particular, a programmer who lacks a complete understanding of a complex system but needs to make an extension can identify an existing feature that is analogous to the planned feature, copy its code, and then modify the copy minimally to achieve the desired effect. Given such a coding style, it is trivial to recognize related code using software tools. The insight here is that similar code sequences should be formatted similarly to make tool recognition easier, perhaps even eschewing changes in indentation and variable names.

3 TOOLS

The information transparency of a design depends on the ability of programmers to apply tools to quickly and reliably recognize implicit design information encoded into the software. Paradoxically, then, the quality of a software design is dependent on the environment in which the software is maintained.

Most environments contain a mix of *lexical* and *syntactic* tools. Lexical tools have recognition powers associated with regular expression matching. Syntactic tools are capable of

context-free parsing and can accurately recognize syntactically recursive structures such as nested expressions or compound statements. *Semantic* tools employ techniques such as data flow analysis or higher-order inference. These tools are typically complicated to build, computationally slow, and require enormous machine resources to apply to an entire program [AG96]. Moreover, the scope of their application tends to be narrow (e.g., a tool employing data flow analysis might be capable of only slicing). Because of these limitations, they are generally reserved for occasional but critical applications of increasing information transparency. Even a semantic tool can benefit from information transparency because the behavior of code is most easily inferred from the names of constructs and the infrastructure used to implement the behavior.

Maintenance programmers can often find a basis for similarity and differentiation even when it is not designed into the program. If every module involved with a design decision achieves information transparency in a different way—compromising a design’s overall transparency—a programmer’s ability to make the necessary inferences will be excellent, although slow. A programmer, then, is looking to improve the time-quality tradeoff of such analyses. A programmer’s inclination, is to seek consistent information transparency and to use fast tools to discover the basis for transparency, if necessary, and then extract the information. Although fast tools are often less powerful than desired, and not knowing the basis for transparency can also slow extraction, several related queries can be performed quickly to acquire the desired information without too many false matches. One approach is to start with an overly general query, whose false matches suggest refinements to the pattern, and so forth. In this way, fast tools permit a programmer to trade time for accuracy by applying (perhaps gradually acquired) knowledge of naming conventions, formatting, and location of code to narrow the query result to just the required elements.

Inferential tools. Much recent work in software engineering tools has been focused on *reengineering*, which concerns redocumenting, redesigning, and in some cases reimplementing an application’s software because of its unmaintainable state. Much of this work relates to information transparency. In particular, a number of techniques and tools are concerned with *inferring* relationships amongst program components because they are no longer readily visible in the code. RMTTool is unique in that it permits the engineer to suggest an architecture. RMTTool takes an engineer’s proposed mapping of an application’s low-level constructs to a higher level conceptual architecture, and provides a qualitative, graphical report of the congruence of the mapping [MNS95]. Consequently, if a user of RMTTool has a sense of the application’s information transparency coding conventions, convergence to a sensible architecture and its mapping to source code can be accelerated, reducing the need for bottom-up inference.

Searching tools. There are two challenges to making a software system’s globally shared design information visible to tools. First, most large systems are implemented in *multiple* languages, implying that a tool must be able to recognize the encodings across those languages. Historically, this has led to a dependence on lexical tools, which are largely insensitive to the syntactic details. Second, because the encodings are *implicit* design information, the tool has no *a priori* way to reliably recognize these encodings. Few tools are flexible in this regard—RMTTool is a notable exception.

Consider `grep`, a tool not designed specifically for programming. Many programmers need to search for references of one identifier near another identifier (e.g., in the same statement), but it could well be on another line. Unfortunately, `grep` is strictly line based. Another common task is specifying a pattern within a word, but `grep`’s definition of a word is fixed and not appropriate for popular scripting languages such as Tcl. Finally, `grep` has a fixed definition of a successful match. In many cases it would be helpful if `grep` were to let a programmers know that there were some near matches that could be of interest. The tool `lsme`, which performs token-level regular expression matching [MN95] suffers from similar problems and makes character-level matching difficult.

The C star diagram planning tool. CStar can help a programmer (a) understand how a program data structure is manipulated, (b) explore possible restructurings to better encapsulate that data structure, (c) record a plan for such a restructuring, and (d) to carry out that restructuring [GCBM96]. It can also be used analogously for carrying out other kinds of global changes. Because the tool parses, its recognition of syntactic structures is precise. Type information is easily extracted (and exploited in constructing “type” star diagrams), and data flow analysis can be used (but is currently not) to provide more meaningful star diagrams.

Although popular with users, the tool has been criticized by for lacking features that relate to information transparency. For example, users have noted inability to control the tool’s node-stacking policy to represent *their* idea of what is similar (e.g., `myprintf` nodes should be stacked with `fprintf` nodes because both perform output). Some users have wished that a star diagram could be built for all code matching a user-specified pattern; currently the supported patterns are *same-variable-as* and *same-type-as*. Indeed, information transparency principles assert that *part* of an identifier’s name or stylized programming can encode the shared use of a design decision that could be visualized in a star diagram.

4 CONCLUSION

Designers are frequently unsuccessful in designing for change using traditional modularity techniques. The *information transparency* modularity technique can improve a designer’s ability to plan for change by revealing the interdependence of widely dispersed program elements that must be changed together to perform a change correctly. Unlike

information hiding, which represents modules with locality and abstraction, information transparency represents modules by similarity and architecture. With these, a programmer can create locality with a software tool, easing change in much the same way traditional modularity does. Information transparency techniques include naming conventions, formatting style, and ordering of code in a file.

Transparency can be increased by better matching tool capabilities and programming style. A tool's sensitivity to implicit design information is predicated on its customizability with respect to how a programmer chooses to encode design information. Moreover, its model of customization must be consonant with the programmer's to avoid awkwardness and unpredictable results. Speed and the ability to handle multiple languages are also important. Because of the differing constraints among the programming languages used in a software system, the conventions adopted may vary from language to language. Ideally, then, a tool should be able to recognize what language is being used (using the file suffix, say) and customize itself appropriately.

Acknowledgments. I would like to thank Andy Gray, Darren Atkinson, Jim Hayes, and Walter Korman for our early discussions on this topic. I would also like to thank David Notkin and Gail Murphy for their insights based on a talk given at the Dagstuhl Conference Center in Germany.

REFERENCES

- [AG96] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, March 1996.
- [Aho80] A. V. Aho. Pattern matching in strings. In R. V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 325–347. Academic Press, New York, 1980.
- [BG94] R. W. Bowdidge and W. G. Griswold. Automated support for encapsulating abstract data types. In *ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 97–110, December 1994.
- [BG97] R. W. Bowdidge and W. G. Griswold. How software tools organize programmer behavior during the task of data encapsulation. *Empirical Software Engineering*, 2(3):221–68, April 1997.
- [BG98] R. W. Bowdidge and W. G. Griswold. Supporting the restructuring of data abstractions through manipulation of a program visualization. *ACM Transactions on Software Engineering and Methodology*, 7(2), April 1998.
- [BL76] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976. Reprinted in M. M. Lehman, L. A. Belady, editors, *Program Evolution: Processes of Software Change*, Ch. 8, APIC Studies in Data Processing No. 27. Academic Press, London, 1985.
- [GCBM96] W. G. Griswold, M. I. Chen, R. W. Bowdidge, and J. D. Morgenthaler. Tool support for planning the restructuring of data abstractions in large systems. In *ACM SIGSOFT '96 Symposium on the Foundations of Software Engineering*, October 1996.
- [GHK97] W. G. Griswold, J. Hayes, and W. Korman. A reengineering case study. Unpublished Experiment, 1997.
- [GN93] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, July 1993.
- [Gra97] A. J. Gray. Development of an unanticipated member of a program family. Masters Thesis, University of California, San Diego, Department of Computer Science and Engineering, October 1997. Technical Report CS97-560.
- [MN95] G. C. Murphy and D. Notkin. Lightweight source model extraction. In *ACM SIGSOFT '95 Symposium on the Foundations of Software Engineering*, pages 116–27, October 1995.
- [MNS95] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *ACM SIGSOFT '95 Symposium on the Foundations of Software Engineering*, pages 18–28, October 1995.
- [Opd92] W. F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Applications Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Dept. of Computer Science, 1992. Technical Report No. 1759.
- [Par96] D. L. Parnas. Why software jewels are rare. *IEEE Computer*, 29(2):57–60, February 1996.
- [PCW84] D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. In *Proceedings of the 7th International Conference on Software Engineering*, pages 408–17, March 1984.
- [RC93] M. B. Rosson and J. M. Carroll. Active programming strategies in reuse. In *ECOOP '93, 7th European Conference on Object-Oriented Programming*, pages 4–20, 1993.