Assessment of Support for Program Understanding

Eun Man Choi
Dept. of Computer Science
Illinois Institute of Technology
Chicago, IL 60616

Abstract

This paper discusses tools for program understanding during the software maintenance phase. The program understanding is crucial to successful maintenance, but it is still poorly supported by analysis-oriented tools. In the light of cognitive studies for program understanding, we assess the existing tools for program understanding, and suggest an approach which facilitates the understanding of complex code during maintenance via the chunking process. During this process programmers recognize the abstract function or meaning of groups of statements and then piece together these chunks to form even larger chunks until the entire code is understood and mapped out. Chunking support can be effective as part of a maintenance toolkit. It lets maintenance personnel control code abstraction and ask many semantic questions about chunks and their relationship to other parts of the code.

1 Introduction

The key activities in software maintenance are understanding existing code, determining where to make changes, assessing the impact of modifications, rebuilding code after alternations, and regression testing to validate changes. It is recognized that the more complete and correct the understanding, the more likely that the modifications will be correct. The consequences of inadequate understanding include the failure of the system, maintenance problems, and the future loss of maintenance credibility and/or business. It is therefore crucial that intentions behind the code are well understood and analyzed.

Brooks[5] identifies essential difficulties in understanding existing software as follows: the complexity of the application domain and internals of the system, invisibility of the structure and principles underlying large systems, and changeability of software. Complexity is one inherent characteristic of the software: large amounts and various kinds of information are intermingled and hinder the program reader with too much detail. Unfortunately, there is no clear presentation of the architecture for programmers to examine. Invisibility of needed information creates difficulties for programmers. As the program evolves in response to the market place, and grows in size and functionality, the initial architecture slowly disintegrates.

Program understanding impacts a maintainer’s productivity. Chapin in [11] showed that effort for program understanding occurs in the beginning of the maintenance cycle. The quality impact of program understanding arises from a programmer’s action upon incorrect or insufficient information. Soloway et al.[25] showed how a lack of programming knowledge manifests itself in poor quality of program understanding when violate rules of programming discourse and the programming plan. Without proper knowledge of the architecture of the system, the maintainer can make mistakes and modify the code incorrectly.

As system are complex, a process of abstraction is fundamental to understanding. Several empirical studies[16][30] on comparing novice and expert programmers show that understanding programs involves assigning meaning to a program text, more meaning than is literally there. Novices cannot understand the entire program in a line-by-line form because of limited short-term memory. However, an expert has the abstraction ability to understand a large program through a process called “Chunking”. In other words, program understanding involves successive recordings of groups of program statements into increasingly higher-level semantic structures. Thus, patterns of understanding are recognized as higher-order chunks; patterns of chunks are recognized as algorithms, and so on. The chunking process plays an important role in the process of understanding. A hierarchical internal semantic representation of the pro-
Program is built during program understanding from the bottom-up. In conclusion, instead of absorbing the program on a character-by-character basis, programmers recognize the function of groups of statements, then piece together chunks to form ever larger chunks until the entire program is understood. Obviously the size and number of chunks depend on a programmer's experience, knowledge, and abstraction capabilities.

von Mayrhauser[28] defines the support to program understanding as consisting of two types: analysis-oriented tools, and code-oriented tools. As it will be shown later, a variety of analysis-oriented tools is available in software maintenance. Analysis-oriented tools usually give information about the execution sequence (control flow) and the transformational effects on data objects (data flow) in graph form. Another approach suggests that a program should be analyzed in a code context and that all analysis information should be more strongly connected with code. This supports understanding better.

The purpose of this paper is to summarize tools supporting program understanding to-date and address the differences between analysis-oriented and code-oriented tools. In order to assess tools or approach, this paper focuses on the behavioral aspects of program understanding and a model of the cognitive process underlying program understanding. Then, we will introduce how to provide tools for a better code-oriented view, that tools are better suited for the mental cognitive process and provide a more controlled analysis view.

2 Tools - State of the Art

Various approaches exist to support program understanding. One traditional approach is to reproduce multiple views from the source code. This provides dependency information in graph or table form. The users navigate through complex interrelationships within program components. Figure 1 presents an overview of our classification of program understanding tools. Here, rather than enumerating all tools and approaches, we classify the area, explain each area consideration to the understanding process, summarize the emphasis of each area and figure out what kind of techniques can be applied to tools that aid the programmer in code maintenance and evolution.

<table>
<thead>
<tr>
<th>Class</th>
<th>Approach</th>
<th>Emphasis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis tools</td>
<td>dependency within program graphical representation of component</td>
<td>multiple views</td>
</tr>
<tr>
<td>Restructuring tools</td>
<td>extract structural properties modules structure of large program</td>
<td></td>
</tr>
<tr>
<td>Hypertext applied tools</td>
<td>provide internal information in code context</td>
<td>binding documents to code</td>
</tr>
<tr>
<td>Reverse engineering tools</td>
<td>extract design information from the code</td>
<td>recover lost design information</td>
</tr>
<tr>
<td>Abstraction tools</td>
<td>suppression of unimportant formal abstraction detail</td>
<td></td>
</tr>
<tr>
<td>Code processing tools</td>
<td>user interface to support code processing</td>
<td>controlled analysis views and checking process</td>
</tr>
<tr>
<td>Knowledge-based tools</td>
<td>extracting program plan automatic documentation</td>
<td>software information system</td>
</tr>
</tbody>
</table>

Figure 1: A classification of tools for program understanding

2.1 Analysis Tools

Analysis tools are powerful means of studying the behavior of programs to determine how programming is done and what programs do. Most analysis tools perform a static analysis that analyzes code without execution. Several analysis tools have been proposed. These tools provide alternate views of the program. We can loosely classify code analysis techniques into the following categories:

- cross reference facilities[17]
- code reformatters, pretty printers[6][18]
- call graph generator[21][22]
- data flow analysis[7][23]
- structure chart[31]

A cross reference listing indicates which variables were declared and which were not, even what their initial values are. And it provides the calling structure of who calls whom and from where, and procedure names and statement numbers where formal parameters, local variables, and global variables are defined, set, and used. This is a valuable aid in capturing the relations between objects of a program.

Pretty printers, widely used to produce neatly formatted versions of program, help program understanding[6]. In order to promote understanding, pretty printers use commands to boldface keywords, italicize identifiers, and list line numbers. However, Oman, and Cook[18] pointed out the problem that source code listings are not really designed to help users read and understand, rather they are designed to be pretty with little regard for improved program comprehension. They claimed that traditional book typographic formats work very well to aid program understanding. The book paradigm uses type style, sentencing, paragraphing, sectional division, chapter division,
prefaces, indexing, and pagination. That gives high-level organizational clues about the code, and low-level organizational chunks and beacons to serve as an efficient form of information. However, it must be limited in handling large complex code because only indexing is provided as an abstraction mechanism.

The call graph generator[21][22] represents the architecture graphically to make it more understandable. Complex interrelationships among program parts are not revealed by textual representation of the code. For example, VIFOR[21] classifies objects(modules, declarations) and relations(belong to, call, reference) in a program. In the visual form, the program is represented by a graph consisting of icons representing modules and lines representing relations. VIFOR also has the abilities to scale down to make more entities and relations visible, or to scale up to provide a clear view of the information displayed.

Data flow analysis[7][23] focuses on the flow of data. It is helpful to know which parts in a program affect the value of the variable. Module level data flow diagrams are often used in maintenance phase as well as design. In the diagram, each node is a module or logically independent unit. Sequencing of actions become clear by showing the flow of data objects. From the point of data flow testing, it can be used in classifying each variable occurrence as being either a definition or a use. The maintainer can recognize which variables affect computation.

A structure chart[31] is a call graph showing the interconnections among procedure/functions. That interconnection is shown by arranging procedures/functions in levels and connecting the levels by arcs. An arc drawn between two modules at successive levels means that at execution time program control is passed from one module to another in a top-down direction. The structure chart shows the architectural property of a large program with a broad point of view.

In many cases, certain relationships among programs or program elements are displayed graphically, but mapping from graph elements to code is not provided to the user. The programmer must navigate through multiple views of the program by himself.

2.2 Restructuring Tools

Restructuring means the transformation from one representation to another at the same relative abstraction level[9]. The term restructuring came into popular use from code-to-code transformations which convert an unstructured program into a structured one(goto-less). However, the term has a broader meaning that recognizes the code, reshapes code structure and recasts the user program with more structural access capability.

Choi and Scacchi[10] suggested restructuring the design of large systems, a kind of reverse engineering technique. They introduced four properties of a system which must be supported by reverse engineering: structural properties, functional properties, dynamic properties, and behavioral properties. They focus on extracting the structural and, to a lesser degree, functional and dynamic properties of large systems - systems composed of modules and subsystems. This is the reverse-engineering of a system-level design description written in a module interconnection language, NuMIL. After an initial design description is generated, the system restructuring algorithm constructs a hierarchy of the system's modules and subsystems.

Even though they did not suggest any user interface for system hierarchical information, extracting a hierarchical design description is very helpful to large program understanding. Program readers would be able to identify subsystems which are composed of other subsystems or modules, and find the module interface through a generated MIL spec. Unfortunately, a good paradigm of restructuring for covering intra and inter module has not appeared. Choi and Scacchi's[10] restructuring scheme is for extracting the macrolevel architecture of large software, while other research intends to guide the microlevel structure of software. Our research is for a single restructuring method both at the module level(inter module) and at the system level(inter module).

2.3 Hypertext

Hypertext is an appealing technology that addresses many of the problems confronting software maintainers. Hypertext systems are primarily intended to support exploration of non-linear text. Hypertext can be thought of as a graph with nodes containing information units and links that explicite relationships between information units. Many view points of hypertext exist in terms of application. When looking for a multimedia user interface, we consider hypertext technology as a great user interface scheme. From a database view point, one can view a hypertext system as a tool that provides nonlinear access to the desired information by viewing, linking, and annotating nodes in a large textual and graphic database. The power of cross-referencing between related components of information and between differing levels of information has been recognized.
Some trial applications of hypertext manage software documentation for maintenance purposes. B. Blum[4] suggested a model of a software maintenance documentation system based upon the hypertext concept. He classified essential information (1) about the application domain that provides an understanding of what the software product is to do, and (2) about the implementation domain that suggests how software can meet the desired needs and how the product may be realized.

Fletton and Munro[14] also discuss the importance of documentation in the maintenance phase and propose a tool based on hypertext technology that can meet a number of the documentation needs of the maintenance programmer. Their introductory paper explained the possibility to document a system from an abstract overview level down to the statement level of source code. If the source code were included within the hypertext document then all the systems documentation could be traced to the actual source code that is described.

The big leader of applying hypertext to maintenance is W. Scacchi[24]. He considered a software system as a fully structured description whose syntax and semantics can be formally defined and automatically analyzed, and weakly structured descriptions whose content can be text-processed. As such, automated mechanisms must be provided to identify and trace relationships across multiple semi-structure descriptions of the same system in order to configure, validate, and maintain the consistency of interrelated software descriptions.

The hypertext approach is an attempt to help program understanding by binding documents and code. If some code segments or code objects(variable, procedure name etc.) are not understood, related document parts can be revealed successively by user selection. Therefore, that approach is not an abstraction for program understanding, but only setting a relationship between deliverables and code. This multimedia approach for binding graphic, text and voice data does not need to be applied to construction of non-linear source code. We need only a few characteristics of hypertext: the chunking concept, intra/inter code traceability, and non-linear structure of text.

2.4 Reverse Engineering

Reverse engineering is the process of analyzing a subject system to identify the system's components and their relationships and create representations of the system in another form or at a higher level of abstraction[12]. The primary purpose of reverse engineering is to increase the overall comprehensibility of the system for maintenance and new development. Therefore, reverse engineering does not involve changing the reverse engineering subject system. It is a process of examination, not a process of change or replication. Many tools for reverse engineering have been developed to gain a sufficient code, design, and requirement level understanding to aid maintenance.

One subarea that is widely referred to is redocumentation. Redocumentation is the creation or revision of a semantically equivalent representation. The resulting forms of representation are usually considered alternate views(for example, dataflow, data structure, and control flow) intended for a human audience. That implies that the intent is to recover some information about the subject system that existed or should have existed. Some common tools aided to perform redocumentation are pretty printers(which display a code listing in an improved form), diagram generators(which create diagrams directly from code), reflecting control flow or code structure, and cross-reference listing generators. The key goal of these tools is to provide easier ways to visualize relationships clearly.

Design recovery[1][2] is also a subset of reverse engineering in which design abstraction is extracted from a combination of code, existing design documentation(if available), personal experience, and general knowledge about the problem and application domain. According to T. Biggerstaff, design recovery must reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth. But, realized design information is only recovered at the architectural level especially needed to understand the larger complex system. Following the preliminary frameworks for reverse engineering presented in [1][2] the process can recover design first, and then continue to recover requirements information. The recovered structure chart is particularly useful in that it shows the module dependencies and the visibility, by direction, of data and variables among modules, thereby helping the designer and maintainer understand the structural properties of a program.

We need to remember Biggerstaff’s comment[3] that a fully automated reengineering system for developing human oriented conceptual abstraction is not feasible. Human understanding is an incremental process that defers precision to gain succinctness in the early stages in order to prune the possible interpretations at each step of the understanding process. Therefore, the tools for step-by-step abstraction are
definitely needed.

2.5 Abstraction

For research on program understanding, Colbrook[12] and Pleszkoch[20] gave the important clue, abstraction, to handle complexity of code. Colbrook insisted that it is necessary to reduce the amount of complexity and it can be achieved by the use of formal abstraction. Abstraction is the process of ignoring certain details in order to simplify the problem. He proposed a technique, based upon stepwise abstraction, for the comprehension of a structured program. He considered program reading as replacing the prime program(program segment with single entry and single exit control flow) with logic comments based upon the functionality of the prime program block. If this logic comment is automatically expressed in a formal manner, it is helpful to program reading.

Pleszkoch et al.[20] proposed the function-theoretic principle of program understanding. According to their research, program abstraction is an emerging technology that permits automatic abstraction of business rules from code to recover precise functional documentation. The principle of program abstraction is the substitution of functions for rules which defines a stepwise abstraction process, in repeatedly substituting function specifications for control structures, beginning at the lowest level, continuing in this manner until no control structures remain to abstract. The process is moving from a structured program at the bottom through successive levels of abstraction to arrive at the function at the top level.

Those two formal abstraction principles gave the crucial idea to support the abstraction of a program in maintenance or re-engineering. Their suggestion is in accord with the results of cognitive research already mentioned. In program understanding, suppression of unimportant details brings complex code under the maintainers' control.

2.6 Code Processing

Recent research on the user interface[11][13][27] observes that the user interface is a key issue in enhancing program understanding. The presentation of information from tools such as static analysis must not complicate, nor distract from, the task of understanding the program. Those three research articles deal with how to improve the user interface for better program understanding.

Cleveland[11] developed a referencing structure between the various static analysis presentations and the source. He focuses on multiple window presentation to allow related pieces of information to be visible simultaneously. He suggested an intelligent cursor that is aware of the type of data to which it is pointing and can thus, when the programmer selects the piece of data, offer the programmer only those options valid for that type of data.

Cordy et al.[13] suggested a source program viewing and editing system, called TuringTool, designed to support maintenance. They observed that source code elision, which means wiping out unimportant code in display, provides a single comprehensive viewing paradigm and will be of great assistance to the programmer in creating and manipulating a program. Since maintenance programmers often find it necessary to view the program source in ways that are fundamentally different from the way it was viewed by the developer, the tool can visualize the program from nonstructural points of view.

Another version of program elision was Expert Dataflow and Static Analysis tool(EDSA), a commercial product[27]. EDSA helps the maintenance programmer analyze and understand the logic of Ada source code while simplifying the reading of the code. A view may be thought of as a pretty printed version of the program with selected syntactic elements elided. With this facility, users have the capability to remove irrelevant details and to focus on important ones. The browsing facilities of EDSA allow users to traverse their programs either textually, following possible control flow paths, or following the data flow structure of the program.

The above code processing research points out the important idea, i.e. suppression of unimportant program segment information. But, they did not support informal information useful to understand a chunk, but only elide some portion of the code. Elision takes part in improving the understanding environment, but it is not a panacea to solve the problem of understanding.

2.7 Knowledge-based Approach

In order to be more helpful to program understanding, recently some new research of code analysis has been tried: a program plan analyzer[15] and automatic program documentation[19]. These approaches start with understanding the programmers’ cognitive process to provide more active help. In both areas, a technique called transformational analysis[19] is used: some semantic properties are recognized in the code and improve program understanding. Almost all techniques can be replaced by descriptions of their goals. The results of this work were integrated...
into our chunking concept. Some of the support from
transformational analysis are summaries of the code
and answers to maintenance questions about it.

Soloway and Ehrlich[25] argue that programming
knowledge is organized into two basic components:
rules of programming discourse and programming
plans. Rules of programming discourse describe stylis-
tic guides to development of programs, guides that
are independent of particular languages or applica-
tions; (for example, “variable names should reflect
their function”, “a variable initialized by an assign-
ment statement should be updated via an assignment
statement”). A programming plan is an abstract
structure containing a decomposition of goals that
must be achieved to accomplish the function of the
program. A program plan links the bottom-level goals
to specific computational structures for performing the
function. Most programs are composed of plans that
have been composed by interleaving lower level plans.

Some program analysis tools[16] take a program as
input plus a knowledge base of the programming plan,
and find instances of plans in the code. They sug-
gested two capabilities that might be worth automating
in a program analysis tool: the ability to summa-
rize a piece of code, and the ability to answer ques-
tions. These same two tasks, summarization and ques-
tion answering, have been used to test understanding
ability in computer programs which understand natu-
ral language stories[20].

Ourston and McBeth[19] proposed an approach for
deriving an English language description of a com-
puter program directly from the source code. They
present two kinds of machine translation: simple state-
ment rendering, to derive the English language equiv-
alent of the program statements, and concept abstrac-
tion, to deduce the purpose of the program. Concept
abstraction involves program understanding, and can
serve as a basis for intelligent query support for pro-
viding relevant documentation.

But the technique for recognizing program abstrac-
tions has limitations. It is potentially computationally
explosive in that the input program is compared to all
patterns. Research applying artificial intelligent tech-
niques to automatic abstraction construct typical pro-
gram plans in a library and match plans against the
input program. Currently, it is unlikely to recognize all
of any reasonably complex program also does not pro-
vide for many standard programming constructs such
as recursion, operations with side-effects, or functional
arguments.

3 The AMT

Understanding code can be done top-down or
bottom-up. In top-down understanding hypotheses
are made about how the code works and then tested.
Documents related to the code such as user manuals,
design documentation etc. can form such hypothe-
ses. Bottom-up understanding starts with the code
and builds successive layers of abstraction from there.
Tools that support bottom-up understanding include
browsers and chunkers.

The Ada Maintenance Toolchest[29] consists of the
following tools and components: User Interface, In-
cremental Parser, Control Flow/Data Flow Analyzer,
Regression Testing Support, Object Modification Sup-
port, Database Configuration Management and Con-
trol, and, as one of the central elements, the Chunker.
The basic philosophy of developing tools are the fol-
lowing.

- a program should be analyzed in a code context
  and provide user interface function that help un-
derstanding(highlight, typos)

- code-oriented views are more user-friendly than
  analysis oriented views

- changes and their effects should be visible within
  user code context

- test information should be identified with user
  code views

AMT strives to provide a question oriented code pro-
cessing. For example, if a user asks “what are the
paths I need to test for this change?”, AMT highlight
statements belonging to paths with an option to sup-
press statements not belong to them. That makes the
code connection more strongly than showing a picture
of a control flow graph.

The contents of the toolchest include three im-
portant components: an incremental parser, a struc-
ture editor with display manager, and other incremen-
tal analysis tools. The incremental parser performs
bottom-up parsing and generate a structure tree that
can be used by other tools. Changing a program by us-
ing the editor invokes the parser to parse the changes
the delta, constructs a new structure subtree, and con-
nects it to make a whole structure tree. Another com-
ponent playing an important role for code processing
is the structure editor. That is a kind of switchboard
between editing and analysis/viewing capabilities. In
order to keep the toolchest extensible, we decided to
separate the display manager from the structure editor. That makes the display manager the traffic controller among many different kinds of information to be displayed. That concept is necessary for extendability of tools.

The current prototype of the AMT includes the following [28].

1. Browser: provides to zoom-in/zoom-out capabilities for syntactic objects (variable, module calls) on display. That is, a high level view must be available that can be traversed and explored via function keys. It also can make the program's structure more visible much like a table of contents for the program. To find an object and understand it, the programmer must rely on many browsing functions, not only paging, line numbering but also keyword searching, wandering from displays of high level code representation to low level code text and searching through the data and control flow of the package, chunk or module.

2. Chunker: provide user defined blocking, restructuring and level of abstraction (in this article we deal with the chunker).

3. Slicer: show interesting paths within a code context. That displays subsets of the original source code based on user specified input criteria. The slicer includes several options for execution slice, path slice, define-use slice, delta slice, data dependency slice, coverage slice, etc.

4. Data flow analysis: performs data flow analysis for chunks, slices, changes, inter/intra module.

5. Metrics: include traditional metrics for complexity measure, delta metrics for measuring change, and metrics to support AMT functions such as percent of code condensed in a chunk or percent of paths tested.

6. Test support: support test paths as slices, assertions or test cases, statistics of test coverage.

7. Notepad: provide a method for the user to note some important point (so called programming plan[25][26]) along with original code during understanding process.

Chunking provides the foundation for the abstraction, restructuring and reverse engineering that is based on programmer understanding as opposed to solely on syntactic and "semantic cut-and-paste". The chunker as the remainder of the toolkit is designed around the concept of code processing[28], incremental analysis for all tools, and aggregation of analysis information from several tools as necessary to answer user questions.

## 4 Supporting Program Understanding via Chunking

Depending on the degree of maintenance experience, a program may be better understood by the information such as domain knowledge, programming plan, or design abstraction. Some of these are impossible to deduce simply from an analysis of the source code. For that reason, there were several approaches such as binding mechanism between code and documents using HyperText or extracting design information. However, those approaches give the program reader only one clue to construct mental knowledge about the program. They do not support the program understanding process.

The primary purpose of chunking is to bind the information to actual code for increasing the overall comprehensibility of the system in maintenance and new development. A chunking definition looks like putting a useful piece of information instead of the code. Through the basic chunking mechanism a programmer marks objects such as a section of code, a variable, a (part of a) data structure, a (part of an) object, etc. with a description that reflects the programmer's understanding of what it is or does (the abstraction).

The chunker provides a useful abstraction mechanism based on the abstraction capabilities of the user. By a simplified description of code that emphasizes some of the details while suppressing others, the chunker can shorten the time to understand a large and complex system. The chunking mechanism is in harmony with the abstraction which is the process of ignoring certain details in order to simplify the problem.

Restructuring is an important motivation for the chunking concept. It means the transformation from one representation form to another at the same relative abstraction level. Typographic arrangement of code or extracting a hierarchical design can contribute to extracting structural property of the code with less effort, while our chunking can both extract and inject structural information.

In basic chunking the user specifies a name and description for a variable, data structure, section of code or comment. These descriptions would be displayed
instead of the actual code. This supports program understanding which attempts to abstract the logical structure of a program.

A chunking unit can be not only a syntactic component—a program block which is sequence of statements set off by keyword pairs like Procedure/End, If/Endif, and Do/End)—but also a dynamic structure which is the basis of compressing the code in a way the programmer understands. For bottom-up understanding, there must be many questions about the chunk: module interface, variable usage, reusability metric. The chunker has several function menus for: define chunk, undefine chunk, zoom in/out, global zoom in/out, show variables, show number of lines in chunk, edit chunk description, show list of chunks.

4.1 Syntactic Chunking

The notions of a chunk can be considered as the unit of comprehension or cognitive decomposition. In that sense, we can try to find the chunking unit automatically. First, a syntactic unit (procedure, if, while block) can be a chunking unit. Syntactic chunking corresponds to the chunk pertaining to the structure and syntax of the program. In other words, source code block will be first identified as candidates for the chunk as a unit of comprehension. For example, maintainer can try to understand complex structures (variables and types) and give a formal (like a specification language) or informal description (natural language) which plays a critically important role in the human understanding of program. Then, internal blocks within a procedure represent broad and global roles of the block. This is an incremental bottom-up process that allows the user to understand the broad global structure of the large complex program concept.

4.2 Dynamic Chunking

Another possible unit of a chunk is a dynamic unit which overlapped between control structures. A chunking unit does not necessarily correspond to a syntactic block. Sometimes, one functional chunk can be overlapped between two syntactic blocks. This concept results from the observation that program abstraction does not always follow the program structure, but one abstraction is spread through a whole module. In particular, Soloway[26] found that the programming plans were realized by lines scattered in different parts of program and programmers had difficulty understanding code with such delocalized plans. Thus, we need a method to handle a dynamic chunk.

The user can mark localized objects, make them invisible, and replace them with an abstract description.

These form the basis for possible restructuring of code which is that the user defined chunk structure. The user can restructure the code and assess how changes affect chunks.

4.3 Semantic Questions

The key objective for tools that support program understanding is to find commands semantically rich enough to give the user detail information and traceability about the chunk. After the user customizes the program structure by information suppression, they need semantic information about chunks to figure out change effect or restructuring. For example, which variables are used in a chunk of code and how? How much the chunk affected by changing functionality? Can this chunk be an independent procedure? Many semantic questions will occur in the change effect analysis. The maintainer can analyze the individual chunk by following semantic questions and developing his/her own abstractions for the program.

- **Understanding** Which variables are used in a chunk? What is the scope of individual variables? What are the important objects related to functionality? Which parts of the system use it? What is the key functionality of a chunk?

- **Ripple Effects** For analyzing change effect, there can be several semantic questions about a chunk. What are the functions/variables directly or indirectly affected by changing the chunk.

- **Interface Extraction** A chunk or program block will generally have a meaning for one simple functionality: it might be a declaration or procedure that execute a certain function. Accordingly, we can imagine an interface extractor which examines variables in a chunk to determine local/global variables. Global variable can be parameters if the chunk is developed into an independent module for reuse or restructuring.

- **Procedure Extraction** Procedure extraction means that the user highlights codes and commands to make this chunk of code a function or procedure. The procedural extraction facility provides a procedure header with parameters extracted from interface extraction.
4.4 Reuse Chunk

Software reuse is the reapplication of various kinds of knowledge about one system to another similar system in order to reduce the effort of development and maintenance of that other system. Several reuse technologies were suggested for finding, understanding, modifying and composing components. But, most research picks up the module as a reusable unit. Modules subtly encode very specific information about a variety of things: parameter or variable initialization, hardware-dependent code, run-time library and so forth. A module is an independent syntactic unit which can be extracted, compiled stand-alone and reused somewhere. But, each one module does not always have one pure unique function but maybe composed of several functions. If the components (chunks) are sufficiently abstract and each captures only one aspect or principle of algorithm, we may find reusable components with relatively small effort and reuse it. Such an abstraction scheme is a critical issue of reuse research.

This set of questions illustrates a trade-off the choice of reusing a particular chunk or not reusing it due to the difficulties involved. Also, the maintenance personnel has to decide which chunks can be reused if many of them are available. In addition, metrics questions help the maintainer to identify a reuse component.

4.5 Metrics

The metrics about a chunk will be helpful to understand the code restructured by chunking. The percent of code condensed in a chunk and percent of code in the overall system will be an example. Metrics can show how many local or global variables are declared/used under the chunk to guide procedure extraction. Besides size and complexity metrics of a chunk the structuring depth of chunking will be a good metric for indicating level of detail.

5 Conclusion

This paper summarized several approaches to support program understanding and suggested an approach to chunking that starts with a “blocking” feature similar to word processor for segments of code and provides functions to analyze and structure such user defined chunks. The chunker supports the understanding process which can recall larger amounts of information by retrieving a single information unit that acts as a cue. Building these higher-level semantic units is an important part of program understanding. This combined with X-Windows viewing capabilities for elements/levels of this structure provides visual aid for program understanding.

Questions related to semantic meaning represent a way of thinking about the kind of tools that can help the maintainer with modification. In particular, these questions focus attention on how to manage the complexity of code by providing tools for better code oriented-views and which are more better suitable for the mental cognitive process and more controlled analysis view. Our perspective of program understanding can expand program maintenance tools.

We can augment reverse engineering and restructuring algorithms to make them work with understanding as defined by the maintenance programmer. Also, program understanding issues are strongly tied to the ability to reuse. Indeed, a way to fix a system for improved understandability is to reuse the program during development or evolution. This way, the resulting structure of the code will parallel how the user understands the software and work with user understandable chunks.

References


