Available online at www.sciencedirect.com
ScienceDirect
Decision Support Systems 42 (2006) 469–491
Decision Support Systems
www.elsevier.com/locate/dsw

Developing maintainable software: The READABLE approach☆

Cecil Eng Huang Chua[a,*], Sandeep Purao[b], Veda C. Storey[c]

[a] Nanyang Business School, Nanyang Technological University, 639798, Singapore
[b] School of Information Sciences and Technology, The Pennsylvania State University, University Park, State College, PA 16802, USA
[c] J. Mack Robinson College of Business, Georgia State University, Atlanta, GA 30303, USA

Available online 13 June 2005

Abstract

Software maintenance is expensive and difficult because software is complex and maintenance requires the understanding of code written by someone else. A prerequisite to maintainability is program understanding, specifically, understanding the control flows between software components. This is especially problematic for emerging software technologies, such as the World Wide Web, because of the lack of formal development practices and because web applications comprise a mix of static and dynamic content. Adequate representations are therefore necessary to facilitate program understanding. This research proposes an approach called READABLE (Readable, Executable, Augmentable Database-Linked Environment) that generates executable, tabular representations that can be used to both understand and manipulate software applications. A controlled laboratory experiment carried out to test the efficacy of the approach demonstrates that the representations significantly enhance program understanding. The results suggest that the approach and the corresponding environment may be useful to alleviate problems associated with the software maintainability of new web applications.

Keywords: Web development; Maintenance; Programming

1. Introduction

Software maintenance remains the most expensive component of information system development, consuming 50% to 80% of resources [7,38,42–44,67,86]. A major contributor to this cost is inadequate program understanding [10,42–44,56,94]. When programs are difficult to understand, maintainers, who are often not the same as developers, spend more time reading code, and make incorrect modifications to it. Most approaches to improving program understanding (e.g., CASE Tools, sophisticated Integrated Development Environments (IDE)) add presentation mechanisms to existing programming languages [11,15,45,52,72] instead of attempting to fundamentally change the way code is written. The understanding of code is thus linked to continued use of the CASE tool or IDE. This can be problematic because these tools are often

☆ A preliminary version of this paper was published in the Proceedings of AMCIS 2003.

* Corresponding author.
E-mail addresses: cchua@cis.gsu.edu (C.E.H. Chua), spurao@ist.psu.edu (S. Purao), vstorey@gsu.edu (V.C. Storey).

0167-9236/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.dss.2005.04.002

used during the initial development of an application but are difficult to employ during maintenance. Maintenance typically involves looking at someone else's code, often without access to the design or rationale captured using a CASE tool or an IDE.

The problem is more acute in applications developed for the World Wide Web, where documentation practices can be especially difficult to follow because of rapid development cycles [2,5]. The challenge is exacerbated by the structure of these applications, which includes a hybrid of static content and programming code [6,13,29]. The hybrid nature of such applications makes understanding the program, and consequently maintenance, an onerous task.

The objective of this research is to: *propose a new approach to authoring applications (especially web applications) to improve their understanding during the maintenance phase*. We conceptualize the solution as an approach that helps the programmer communicate to the maintainer information about a key program understanding construct, i.e., control flows [56]. To facilitate this communication, the information must be easy for a programmer to capture, and a maintainer to understand. The contribution of this research is an approach for capturing such manipulable representations of control flow information, and a corresponding environment that generates executable code based on these representations. Our work builds on prior research on control flow representations such as state-transitions [34,35], Turing Machines [78], and tables [53,58] to propose a layered approach, called READABLE (Readable, Executable, Augmented Database-Linked Environment), that represents the program components, their control flows, and data relationships as tables. To demonstrate the feasibility of the approach, we extend the Java programming language to incorporate READABLE constructs and test its efficacy with a laboratory experiment. The results indicate that READABLE improves maintainers' program understanding, thereby improving software maintainability.

The remainder of this paper proceeds as follows. Section 2 presents related research. This is followed by a discussion of the foundations of the READABLE approach. A prototype of READABLE is described in Section 3, along with an illustration that highlights the concepts implemented. In Section 4, we develop hypotheses to evaluate the efficacy of the approach, and describe a laboratory experiment conducted for this evaluation. A discussion of the results with implications for practice and research is found in Section 5. Section 6 concludes the paper and identifies avenues for future research.

2. Prior research

2.1. Program understanding

Program understanding deals with how maintainers understand code written by someone else for the purpose of ongoing application maintenance. Often, maintainers must read the code, without access to complete specifications, and without recourse to the rationale. As a result, as much as 50% of a maintainer's time is spent reviewing existing code [92]. Research on program understanding, therefore, relates human psychology and cognition to software maintenance. Constructs such as schemas [8,63], short-term memory [48], and chunking [84] are employed to explain and predict maintainer behaviors [23,89], and thereby recommend methods for training them [48,57]. This research suggests that the appropriate mindset for program understanding involves searching [31] and problem solving [1], not reading a standard text document. When maintainers read programming code, they jump around instead of reading the code from top to bottom [82,83].

One promising approach to improving program understanding is to represent code as tables. Parnas et al. demonstrate that software specified as a set of tables can be both simultaneously easy to understand, and useful for rapid prototyping [53,58]. Kimbrough and Yang [39] suggests that tables provide a natural method for communicating design specifications. Tables provide a method to easily organize disparate sources of information, thereby encouraging rapid comprehension [20,46,87]. Although some find alternate representations (e.g., graphs) superior for particular tasks [69,80], all agree that tables often convey structured information in a clearer way than text [55,85]. Tables have also been employed as representations for numerous technical tasks including the representation of mathematical models [28], and specific decision support systems [75].

Tabular representations can also be adopted to alleviate what Pizka [59] argues is the greatest problem in software maintenance: the separation between the model, code and execution system. For example, the written code must be compiled before it can be used. As a result, changes to the code are not necessarily reflected in changes to the system. Attempts to preserve this relationship have been attempted, for example, in tools or frameworks like Eclipse [12]. However, such tools and frameworks do not establish a 1:1 relationship between code and the system. Instead, code changes not reflected in the model are marked as "protected" so that when the model is revised, the modeling tool does not overwrite it with new code.

To further unravel what maintainers must understand, Pennington [56] proposes four constructs: (1) function, (2) control flow, (3) data flow, and (4) program states. The *function* of the program refers to its intended goal and can be subdivided into subfunctions. The *control flow* of a program identifies how different components of the program interact. The *data flow* identifies how data variables are transferred across components. Finally, the *program states* identify constraints imposed by data values on the execution of components.

These four constructs provide software maintainers with different approaches to understanding programs [89], depending upon their knowledge of the application domain [62]. Those unfamiliar with the domain first attempt to understand its control flow [56], which is then used as a framework for understanding the program. For example, a maintainer unfamiliar with number guessing games may trace through source code beginning with the main module to understand the relationships between the user's guessed number, and the random number generated by the computer. Those familiar with the domain search for code or variables that represent required features [71]. The code or variables are then used as beacons [10,61], i.e., recognizable landmarks in code that allow navigating to other, unfamiliar, parts of the source code [10]. For example, maintainers familiar with guessing game applications may search for components with names such as 'guessnumber()' or variables such as 'mynumber'. In both cases, the control flow construct represents the most important piece of information [73,90]. In the first case, developers trace it from the initial state; in the second, they trace it from the beacons.

Various modeling tools have been developed to simplify program understanding. For example Paakki et al. [52] describe a tool to navigate C code using Hypertext. Others describe tools such as source repositories that allow storage and retrieval of code from a structured database [15,45,52,72]. Such tools are useful to maintainers because they simplify managing the storage and access to code files. They do not, however, help the maintainers understand the code nor do they help developers in creating code that may be easier to understand. Other tools, such as code generation tools (often embedded in programming environments or CASE tools) are useful to developers for generating code. This set of tools work by hiding the complexities inherent in the source code. While they are useful for developers, they can make maintenance more difficult when the hidden code contains a bug or flaw. Yet another class of tools may be identified as modeling and programming environments such as Eclipse (from IBM) and Whitehorse (from Microsoft). These contain yet another set of capabilities that allow model-driven development similar to that embedded in CASE tools such as Rational Rose. Each of these classes of tools provides supporting capabilities for the core task of programming. None, however, addresses the core task of creating the code itself with a view to making the code more easily understandable for maintainers.

2.2. Software documentation

While research on program understanding is useful for improving software documentation, such research assumes a "best-case" scenario where documentation is readily available. However, more often, developers fail to provide useful, and current documentation [27,88]. Research on improving software documentation practices attempts to resolve such issues and can be grouped into three categories: (1) literate programming, (2) reverse engineering, and (3) application generators.

2.2.1. Literate Programming

In literate programming, the developer creates code that other human beings (as opposed to a computer) can understand. This code is then typeset for human consumption and compiled for computer consumption by automated tools. The classic example of literate

programming is WEB[1] [40], although there are other works in this field such as Elucidative Scheme [50]. The main limitation of literate programming is its focus on documenting individual software components. Little attention is paid to documenting relationships between components (e.g., control flow), which is often more important for program understanding [90,93].

2.2.2. Reverse Engineering

Automated tools have been devised to create documentation from existing source code. Examples include JavaDoc [74] and its extensions [32,64], which generate documentation from Java source or byte code, KES (Knowledge Extraction System), which reverse-engineers an ER diagram from a relational database [16], and systems that reengineer documentation from legacy code [77,81]. Reverse engineering tools, however, tend to produce too much documentation [3,91] because all constructs of a prescribed form are extracted, including many that are not relevant to the maintainer.

2.2.3. Application Generators

Application generators are CASE tools that generate complete programs, or program stubs from documentation. Examples include Rational Rose [60], MetaEdit [70] and YACC [37]. CASE tools facilitate documentation, but are not themselves documents. Often, the adequacy of CASE tool documents depends on the expertise and willingness of the original CASE tool user (i.e., the developer) to use the tool. Furthermore, although application generators and CASE tools are useful, they are frequently employed only at the beginning of an implementation [41]. Developers often discover problems with code developed from documentation, and change it, without changing the documentation [27].

The above sub-streams address distinct aspects of software documentation. The first aims at creating code that simultaneously serves as documentation appropriate for human consumption, the second extracts documentation from existing code, and the third uses documentation as the starting point for generating code. None, however, provides an ongoing solution to generate documentation, nor adequately deals with the control flow construct. The problems are even more acute for emerging applications such as ones developed for the web.

2.3. Software documentation in emerging applications

Web applications, especially decision support systems (DSS) [68] pose a special challenge for program understanding for two reasons. First, the process for creating web applications has evolved as new uses for the web were identified. These processes were not considered when the web was first conceived [6]. Thus web applications integrate a number of distinct technologies that follow separate development paradigms [13]. The original technologies of the web (HTTP and HTML) are based on a document markup paradigm [6]. These are coupled to (for example) Java Servlets that follow the Turing machine paradigm of computation, and scripts that adopt a hybrid paradigm. In contrast, traditional applications are based wholly on the Turing machine paradigm of computation [78,79]. Various connectivity standards such as GET/POST, the Simple Object Access Protocol (SOAP), Remote Procedure Calls (RPCs), and simple hyperlinks are employed to connect diverse technologies on the web. Web developers must therefore understand how a bewildering array of distinct technologies interoperate as part of a web application. In some cases, separate technologies employed for disparate purposes are combined within the same object. For example, an HTML page may contain Javascript control code thereby combining presentation and program logic. As many of these technologies are fundamental to the existing web infrastructure, it is likely that this conflict between paradigms will continue into the foreseeable future. Thus, program understanding of web applications is an especially difficult task, because the pieces of information needed to understand the program are all dispersed in many locations (e.g. code components, static content pages, scripts in a page, access to a database among others). A maintainer must understand how pieces of an application from different locations work in concert often in the absence of any documentation from the developers about how these pieces fit together.

Various attempts have been made to solve the multiple paradigm problem. Languages and frameworks such as MAWL [4],<bigwig>[9], and Struts

[1] The name "WEB" refers to the complex relationships between elements in software code, and does not represent an acronym.

[14] have been proposed that provide a single web development framework. These languages and frameworks bring web application development under a single banner but do not provide mechanisms for aligning system documentation with source code. This causes a host of maintenance errors. For example, a maintainer could correct, or compile an incorrect version of the source code, or waste time tracing a problem that has been fixed in the source, but has not been updated to the executable [59].

Second, the incentive to document code is greatly reduced for application development under these web platforms. Perceptions of time are shortened on the Internet because software is delivered in successive versions with a frequency higher than in traditional domains [25]. As a result, software is often developed in an unsystematic manner, and left undocumented [2,5] making it difficult for eventual maintainers to understand applications. Web development documentation tools and methodologies such as Araneus [49], Rational Rose [60], Strudel [26], and OOHDM [65,66] have been proposed for documenting or reverse-engineering web applications. These also suffer from the same limitations as the traditional software documentation technologies described above. Some methodologies such as eXtreme Programming have been developed to address such high speed development. In eXtreme programming, the exchange of tacit knowledge between programmers substitutes for the development of formal documentation [54]. However, such approaches do not consider situations where a web application maintainer is not the original developer.

This review of prior work suggests several important directions for addressing our research objective. First, web application development approaches such as MAWL [4] suggest that it is possible to improve maintainability by proposing an alternative technological platform for web application development. Second, the most important construct for program understanding is control flow [56,73]. Thus, useful approaches for facilitating program understanding must focus on this construct. Third, tables are an appropriate format for documenting software [39,53,58]. Finally, Pizka's argument that code and execution should be the same [59] suggests that a web programming language should simultaneously make program understanding easier, be executable, and serve as documentation. These directly contribute to the approach and corresponding environment we develop in response to our research objective.

Our approach, therefore, shares some of the characteristics of iterative development, round-trip engineering, and research related to automated software development with CASE tools. Our intention, however, is different from the ones espoused in the above streams. We focus on the communication between the developer, who creates the application, and the maintainer, who maintains the application often without recourse to the developer or the developer's work products. Existing CASE tools and IDEs, are well suited for iterative development, and communication within the team of developers during the development process. However, they do not support the extension of software development and documentation practices to ongoing maintenance notwithstanding the promise of round-trip engineering.[2] This is particularly true for web applications, with their multi-paradigm nature. Our research, thus, falls in this middle-ground neglected by research on both software development and software maintenance.

3. Developing READABLE software

We propose an *approach* and *execution environment* for constructing, maintaining and executing READABLE software (*R*eadable, *E*xecutable, *A*ugmentable *DA*taBase-*L*inked *E*nvironment). The *approach* facilitates the writing of software in a manner that separates and explicitly documents the control flow among program components [56]. This becomes part of the communication between the developer and the maintainer, thereby contributing to improved program understanding. The control flow is represented in a separate layer called the application structure. This layer uses the state transition formalism [34] as the underlying documentation mechanism. The control flow remains an integral part of the software, and is converted into executable code by the accompanying execution environment.

[2] Separation between an application and its documentation often confounds the round-trip engineering processes [24]. As a result, round-trip engineering is known to produce unsatisfactory documentation and code [36,51].

The *execution environment* presents a specific instantiation of how the READABLE approach can be implemented. It presents a concrete example of the READABLE approach that can be used to develop testable applications. It uses a tabular representation of the application structure to support generation, manipulation and execution of software. Together, the approach and environment allow the developer to create software that facilitates program understanding for the maintainer. Use of the READABLE approach does not require significant adjustments to work practices, and is compatible with existing tools and practices such as CASE tools and round-trip engineering with the additional convenience of overlapping code and documentation. We describe the approach and environment in more detail below.

Both, the approach and environment are generic, and can be used for developing a wide variety of applications. They are especially useful for developing web applications, because they facilitate integration of diverse technologies in a self-documenting way.

3.1. The READABLE approach

The READABLE approach requires explicit representation of the four programming constructs suggested by Pennington [56]. The application structure layer, thus, contains four constructs: (1) control flow, (2) components, (3) variables, and (4) program states (see Fig. 1).

The *control flow* documents potential transitions between program components. Each transition is specified as a pair of components and a condition that dictates the circumstances under which one component transitions to the other. These potential transitions capture dependencies among components, which, strung together, result in different programming constructs such as sequence, selection, or iteration, depending upon the conditions specified. Sequence is represented by labeling the first component as the previous component, the second as the next component, and not specifying any condition. Selection is represented by specifying the condition as variables and values, which allow the sequence to occur. Iteration is represented by specifying a next component as the initial component of a previously executed sequence. The following notation captures the control flow.

$\text{ControlFlow}_{\boldsymbol{mij}}$ Control flow$_m$ between components i and j

$\text{ControlFlow}_{\boldsymbol{mij}}.\text{Variable}_k.\text{value}$ Transition condition specified as the value of variable k

The *component* construct documents various types of software components including static content, server pages, pages with client scripts, servlets or others. The following information is maintained about each component, Component_i, $\text{Component}_j \in$ Components.

$\text{Component}_{\boldsymbol{i}}.\text{name}$ Name of component i

$\text{Component}_{\boldsymbol{i}}.\text{mode}$ Mode of component i {0=waiting for user, 1=otherwise}

$\text{Component}_{\boldsymbol{i}}.\text{location}$ Location of component i specified as URI

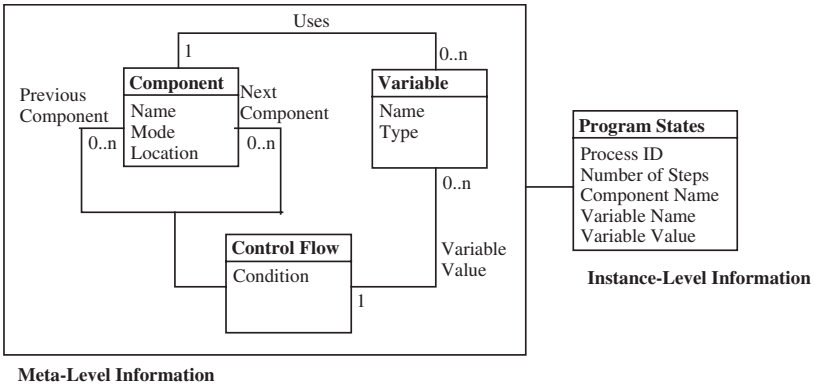


Fig. 1. Application structure layer meta-model.

The *variables* construct captures the existence, and use of the variables, $\text{Variable}_k \in \text{Variables}$, by different components. A variable in READABLE captures anything that can assume multiple values. This includes traditional variables like integers or strings, and more complex data types like states. The notation to capture this construct includes the following.

$\text{Variable}_k.\text{name}$ Name of variable k
$\text{Variable}_k.\text{type}$ Data type of variable k
Use_{ik} Component i uses variable k

The final construct, *program state*, unlike the first three, represents instance-level information, capturing the current state and transitions across states for a given instance of a software. In effect, it traces the activity of the software from the time it begins execution to the time it terminates. The program states construct, therefore, contains instantiations from the other constructs for each unique instance, and may be understood as a 5-tuple consisting of:

Process Id A unique identifier for the process
NumSteps Number of state transitions that have occurred so far
$\text{Component}_i.\text{Name}$ Name of the component currently being executed
$\text{Variable}_k.\text{Name}$ Name of the variable being logged
Value Value of the variable before execution

A team of developers can build software components (e.g. static content, server-side programs such as servlets, client-side programs such as JSP/ASP pages, web services using WSDL) using traditional software libraries (e.g., Java class libraries). The components and dependencies among them are then identified and represented in the application structure layer. The READABLE approach, thus, separates the task of building components from the task of connecting them together. For example, in traditional HTML forms, the URL of the next class or form to be accessed must be specified in the HTML code. Thus, the name of the class file must be known a priori. In READABLE, the HTML code is built without specifying links across components. These are captured and represented separately in the application structure layer, which can be used by the execution environment to run the software and the maintainers to maintain the software.

3.2. An illustration

To illustrate the READABLE approach, we present an example for the well-known number guessing game[3] that has been implemented in the READABLE execution environment at http://readable.eci.gsu.edu:8080/examples/servlet/demo. The example uses the table representation of the execution environment. Other representations (e.g., in XML) are possible. However, the table representation is used because it is easier to understand.

The example implementation contains the following components: Genrand (to generate a random number), DisplayGuess (to display a screen prompting for a number from the user), IncNoGuess (to increment the number of guesses attempted), GoodGuess (to tell the user the guess was correct), LowGuess (to tell the user that the guess was low), and HighGuess (to tell the user that the guess was high). Tables 1–4 represent the constructs shown in the meta-model (Fig. 1). Table 1 demonstrates the control flow, Table 2 the components, Table 3 the variables, and Table 4 the program states.

In Table 1, the branching from IncNoGuess to three possible components (rows 4–6) demonstrates *selection*. If the guessed number matches the answer, the application should tell the user that the guess was correct (GoodGuess). If the guess was lower or higher, the application should so inform the user (LowGuess and HighGuess, respectively). The functions Var(guess) and Var(answer) in the condition field identifies to the application that guess and answer are variables, not commands. The transition from LowGuess to DisplayGuess (rows 7, 3, 5) demonstrates iteration. When the user guesses low, the application increments count and asks the user to guess again. Note how conditionals and looping are specified explicitly outside the web components, making them easier to identify, manage and change than in

[3] The program allows a user to guess a computer-generated number between 1 and 10. For every guess, the computer informs the player whether the guess was 'high', 'low', or 'correct' until the game ends with a correct guess. This game is used as an example for many other prototypes (e.g., MAWL [4]).

Table 1
Control flow

Record	Previous component	Next component	Condition
1	Begin	Genrand	True
2	Genrand	DisplayGuess	True
3	DisplayGuess	IncNoGuess	True
4	IncNoGuess	GoodGuess	Var(guess)=Var(Answer)
5	IncNoGuess	LowGuess	Var(guess)<Var(Answer)
6	IncNoGuess	HighGuess	Var(guess)>Var(Answer)
7	LowGuess	DisplayGuess	True
8	HighGuess	DisplayGuess	True
9	GoodGuess	Genrand	Var(Submit)=Yes
10	GoodGuess	End	Var(Submit)=No

traditional web applications where components can be (for example) identified in <Form Action>commands, in URLs in an HTML form, in a Java servlet, or in an 'include'/ 'import' statement. It is not necessary for the URL to be static following the READABLE approach as long as the placeholder for the URL contains a mechanism to locate the actual URL, e.g., pointer to a controller, which calls the relevant web page, servlet, or other module.

Table 2 shows how the components construct can be populated for the sample number guessing game. Explicitly specifying the components and their URIs in this way greatly enhances a maintainer's understanding of the web application. Web applications have numerous components with URIs that may be dispersed across multiple servers. The component table documents their disparate locations. Note that the "server" specified in Table 2 refers to a logical server. For example, a server pointing to www.ibm.com, could be rerouted to any physical server depending on the physical servers' individual loads.

Table 3 demonstrates variables for the components DisplayGuess and IncNoGuess. These include four

Table 2
Components

Name	Mode	Location
DisplayGuess	Yes	<server>/<directory>/Demo_displayguess
Genrand	No	<server>/<directory>/Demo_genrand
GoodGuess	Yes	<server>/<directory>/Demo_Goodguess
HighGuess	No	<server>/<directory>/Demo_guesshigh
IncNoGuess	No	<server>/<directory>/Demo_incnoguess
LowGuess	No	<server>/<directory>/Demo_guesslow

Table 3
Variables

Component name	Variable name	Data type
DisplayGuess	Limit	Integer
DisplayGuess	NoGuess	Integer
DisplayGuess	Guess	Integer
DisplayGuess	GuessStatus	String
IncNoGuess	NoGuess	Integer

that the first component uses: (1) Limit, a variable that identifies the maximum number to guess, (2) NoGuess, the number of times the user has attempted to guess the answer, (3) Guess, the last guess the user made, and (4) GuessStatus, a string identifying whether the last guess was low or high.

Tables 1–3 describe the application program at rest and are executable code. Changing the values in these tables alters the way the application runs. The fourth table, Program States, captures the manner in which distinct users may navigate through the program. It, thus, captures the dynamic aspect of the application in the form of a log. An excerpt of such a log is shown in Table 4 (a record number is included to facilitate description). This excerpt begins with the fourth executed component, LowGuess (records 1–5), and shows its input memory contents. For instance, the log shows that a user had previously guessed 3, as opposed to the correct answer 4. LowGuess has modified one variable, 'GuessStatus', changing it from a blank to 'Low'. The result of LowGuess is presented as the input to the fifth executed component, DisplayGuess (records 6–10). In the event of an abnormal termination, this enables a software engineer to trace the cause of the termination.

Table 4
Program state

Process ID	Sequence	Component name	Variable name	Value
1	4	LowGuess	Guess	3
1	4	LowGuess	Answer	4
1	4	LowGuess	GuessStatus	''
1	4	LowGuess	Limit	10
1	4	LowGuess	NoGuess	1
1	5	DisplayGuess	Guess	4
1	5	DisplayGuess	Answer	4
1	5	DisplayGuess	GuessStatus	Low
1	5	DisplayGuess	Limit	10
1	5	DisplayGuess	NoGuess	1

3.3. The READABLE execution environment

The READABLE execution environment is conceptualized as a controller that operates on the application structure layer that in turn invokes the web components. The application structure layer comprises the four tables presented in Section 3.2. The controller reads these four tables in the same way a program interpreter or virtual machine reads source/byte code (i.e., as a universal Turing machine [47,78,79]). The READABLE controller identifies the location of the relevant component, and instructs the interpreter to execute the component as needed. If a component requires human intervention (e.g., it is a web form), the controller halts and alerts the user. Once a component is executed, the controller identifies the next component to execute based on the control flow relationships stored in the application structure layer. Fig. 2(b) presents the READABLE execution environment.

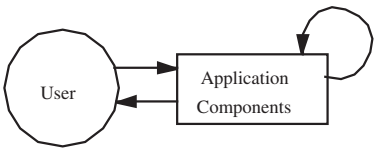
During a typical execution sequence, the controller begins by searching for the first component to be executed (e.g. marked with a reserved name such as 'Begin'). The controller marks it as the 'current' component, and invokes it, passing to the current component any required variables. After the current component is executed, the controller again takes over. The controller compares the updated state information with the transition conditions that initiate with the current component. If a transition condition matches its current state, the controller follows that dependency to identify the next component to be executed. This component is considered as the new current component, and the cycle repeats until the last component (e.g. marked with a reserved name such as 'End') is reached. Thus, as Fig. 2 demonstrates, software with the READABLE approach maintains a clear separation between components and structure in contrast to the traditional web approach.

Unlike controllers in the Model-View-Controller (MVC) architecture [30], or Jakarta Struts [14], a READABLE controller consists entirely of generic code that implements a universal Turing Machine. The READABLE controller employed for any application is, thus, identical. The application itself is specified by entries in the application structure layer, which include the components called, the conditions under which they are invoked and the current values. Changes to the application are accomplished by modifying the application structure layer—not the controller.

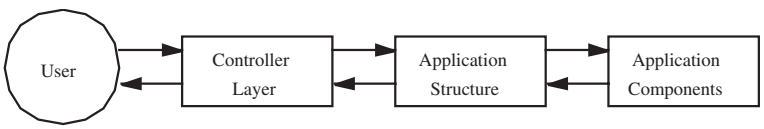
3.4. Establishing feasibility

To demonstrate its feasibility, the READABLE environment was implemented as an extension to the Java language. The implementation of READABLE Java employs the Java Class loader (i.e., the Class 'class,' and the function 'class.newInstance') to load the modules, pass-by-name referencing and hidden form fields to pass variables to HTML forms. The application structure layer was implemented in a relational database management system and the controller was integrated with the Java runtime environment.

An example code snippet that demonstrates how programs can be written with READABLE Java and extensions to HTML to realize the READABLE con-



(a) A Traditional Application



(b) A READABLE Application

Fig. 2. Contrasting a traditional web application with READABLE.

structs is shown in Fig. 3. The code snippet extracts a value "*x*" from a form called "form_1". If "*x*"< some value, then "class1" is called, else, "class2" is called. "HomeURL" refers to the URL of the application's home page, which in the case of READABLE is the URL of the controller. $dump is a command that extends an HTML web page with READABLE control commands. Fig. 3(a) contrasts the HTML code necessary to present the form under the two approaches. Fig. 3(b) contrasts the Java code against the equivalent READABLE control flow table. Syntactic requirements of Java that do not enhance the example are either presented in [square brackets], or are commented by ellipses (. . .).

The following are some of the key distinctions between the two examples. In traditional web development (e.g., the MVC architecture and Struts), the developer explicitly identifies the URL of the component in the HTML form (Fig. 3(a)). However, in READABLE, the developer specifies the URL of the controller layer. Thus, in the example provided in Section 3.2, all HTML forms refer to http://readable.eci.gsu.edu:8080/examples/servlet/demo, regardless of the component being accessed.

Similarly, in traditional web development, the developer would write code to specifically handle a conditional branch (Fig. 3(b)). In the READABLE approach, conditional branching is identified in the control flow table. Thus, with the READABLE approach, the module "thismodule" (and all others like it) are not needed.

The complete implementation of the READABLE execution environment along with further documentation is available at http://readable.eci.gsu.edu:8080/examples/readable.zip. In addition to the illustration above (chosen because of its customary usage in similar research), the READABLE approach has been used to develop several applications, which have been deployed with the READABLE environment on Microsoft Internet Information Server and Apache Tomcat. Two of these are described below.

- *The IS Bibliographic Repository* [17]: This application stores bibliographic information about academic journals in information systems. Users can search and export citations to formats such as Endnote or BibTeX, and can perform bibliometric analyses such as comparing frequency of publication. The IS Bibliographic Repository is available at http://readable.eci.gsu.edu:8080/examples/servlets/isbib.
- *TheAutomated Software Development Environment for Information Retrieval* [18]: This application provides a framework for synthesizing new search engines that incorporate features from existing ones. It can be found at http://readable.eci.gsu.edu:8080/examples/servlet/sf.

Traditional

```
<Form Action="thismodule">
<Input Type="Text" Name="X">
<Input Type="Submit"
Name="Submit" Value="Submit">
</Form>
```

Readable

```
<Form Action="homeURL">
$dump{}
<Input Type="Text" Name="X">
<Input Type="Submit"
Name="Submit" Value="Submit">
</Form>
```

(a) HTML Code

Traditional

```
import class1;
import class2;
  ...
class thismodule{
  [extract x from HTTPServletRequest]
  if (x<[some value])
   [instantiate and execute class1];
  else
   [instantiate and execute class2];
  ...}
```

Readable

Control Flow Table Fragment:

Previous Component	Next Component	Condition
Form_1	class1	x<(some value)
Form_2	class2	x>=(some value)

(b) Java Code vs. Application Structure

Fig. 3. An example of READABLE code.

4. Empirical analysis

To investigate whether the proposed approach with its instantiation as tabular representations [53,58] of Pennington's constructs [56] would contribute to program understanding, a testable empirical model was developed. The model focused on investigating whether programs created with the READABLE approach and executed with the READABLE environment would be more understandable as compared to traditional programs. To reflect the critical nature of the control flow construct in program understanding [56], the model specifically focused on testing improvements in the understanding of this construct without compromising understanding of the other constructs, that is, components, variables, and program states [56]. Fig. 4 presents the research model.

A laboratory experiment was designed to test the claims embedded in the testable empirical model, specifically, the claim that the READABLE approach would enhance program understanding compared to traditional applications, and thus, simplify maintenance. The programming constructs were operationalized in the software following either A1: the READABLE approach, or A2: the Traditional approach. Four hypotheses were formulated that mapped directly to each of Pennington's four constructs (see Table 5). A factual 20-item true/false questionnaire (presented as Appendix A.3) was employed to test the four hypotheses.

Of these four hypotheses, only H1 was of specific interest as it tested for hypothesized benefits of READABLE. The remaining three hypotheses were included as forms of statistical control. Specifically, H1 tested if the understanding of READABLE's control flow would be superior to that of a traditional programming language. As program states combine an understanding of control and data flow, it was also hypothesized that subjects would more easily understand READABLE program states (H2). Even if READABLE improved subjects' understanding of control flow, READABLE would not improve program understanding if it reduced understanding of a program's data flow. Given sufficient statistical power, a failure to reject H3 would suggest that the READABLE approach did not produce unnecessary effects on data flow. Finally, as both READABLE and traditional programming modules are fundamentally the same, we would expect that our treatment and control groups would understand the modules equally well. If a difference was discovered in H4, this would suggest that some extraneous factor was influencing our experiment. Thus, our expectation was that H1 and H2 would be statistically significant in favor of READABLE, and that H3 and H4 would not be statistically significant towards the non-READABLE treatment. Furthermore, H3 and H4 would have sufficient statistical power that they could be accepted.

Table 5
Hypotheses posited

Hypotheses		Test of
H1	Subjects would find it easier to understand *control flow* with READABLE than with a traditional programming language.	Significance
H2	Subjects would find it easier to understand *program states* with READABLE than with a traditional programming language.	Significance
H3	Subjects would not find it any more difficult to understand *data flow* with READABLE than with a traditional programming language.	Power
H4	Subjects' understanding of *modules* with READABLE and modules written in the traditional language would not be significantly different.	Power

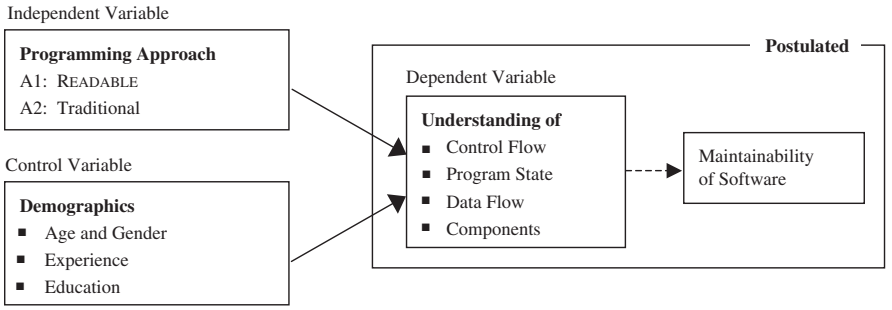


Fig. 4. Testable empirical model.

4.1. Experimental procedure

The study followed a laboratory experiment format patterned after Pennington [56]. Subjects were presented with the scenario that they were hired to maintain code while the chief programmer was on vacation. This required them to understand the source code of the program fully without recourse to an expert. The program they needed to understand was a Java program that solved the *N*-Queens problem.[4] A traditional programming problem was given in lieu of a web-based one, for two reasons. First, we could not identify a "classic" web-based problem to administer to subjects. Second, to control for experimenter/subject relationships, subjects were students that were not taking a course taught by the experimenters. The experimenters, therefore, had to impose on the class time of colleagues. As web-based problems are harder to understand, the experimenters would have had to allocate more time to the experiment, thereby imposing additional burdens on colleagues and students.

The task assigned to the control group was to understand the program written in Java without the READABLE extensions. The task assigned to the treatment group was to understand the same program written in Java with the READABLE extensions. The treatment group had never previously encountered code written in the READABLE style. Therefore, they received a 1-page description of the READABLE approach prior to performing the task (shown in Appendix A.1). To reduce experimenter bias, the subjects were not given any specific training about the READABLE approach. Further details of the experiment are available at http://readable.eci.gsu.edu:8080/examples/readableappendix.doc.

The subjects' understanding of the programs was tested using a 20-item questionnaire, administered under examination conditions, which prevented subjects from discovering the number and type of alternate treatments, and thereby controlled for diffusion of treatments, compensatory rivalry, compensatory equalization, and resentful demoralization [21]. A demographic questionnaire (presented as Appendix A.2) was also administered to test both for equality between groups on extraneous factors, and to test for non-respondent bias. To minimize the effect of guessing, each item on the questionnaire was awarded 1, 0 or −1 points for every correct, blank, or incorrect answer, respectively. Thus, the expected value of guessing on all items was equal to the expected value of a blank questionnaire. The instructions informed subjects that they had 15 min to understand the code. However, the subjects were free to examine the code while answering the questions, i.e., there was no enforced time constraint.

Five true/false items were formulated to test each hypothesis. Examples of these 20 items and the constructs they mapped to are:

H1. *Control Flow*—All control-flow questions required that subjects identify possible order sequences between two modules. To answer control flow items correctly, a subject would have to trace through the linkages between modules. For example, 'add_solution() can be called after solution()'. Items 1, 5, 9, 13, and 17 were control flow items (see Appendix 3).

H2. *Program States*—The program-state questions required that subjects identify possible or actual values of variables that result from execution of the program. Each question in this set links the value of one or more variables to a module. To answer program state items correctly, a subject would have to work through the control flow to evaluate values of variables that reflect program states. For example, 'the minimum possible BoardSize is 0.' Items 3, 7, 11, 15, and 19 were program state items (see Appendix 3).

H3. *Data Flow*—The data flow questions tested knowledge of relationships between data elements of separate modules. For each question, a variable in one module would be linked to a variable in another. To answer data flow items correctly, a subject would have to work through the module invocations to understand what data is passed across modules. For example, 'the array returned by solve_queens() becomes cur_sol.' Items 2, 6, 10, 14,and 18 were data flow items (see Appendix 3).

H4. *Modules*—The module questions tested knowledge about the processes within the modules. To answer a module question correctly, the subject

[4] *N*-Queens is a classic mathematical problem where one places *N* chess queens on an $N \times N$ board so that no two queens can take each other. The problem was selected, because of its clean control flow characteristics. For a good illustration of how *N*-queens is solved, see http://www.math.utah.edu/~alfeld/queens/queens.html.

would have to refer only to the module in question. For example, 'when the solve_queens() function exits, curline is always equal to boardsize.' Items 4, 8, 12, 16, and 20 were module items (see Appendix 3).

A learning bias towards a web-oriented programming language could not be controlled for because the study could only be administered on subjects having some programming knowledge. All subjects had been instructed in the programming language (i.e., Java), but without any READABLE constructs. As the experiment was designed to test whether READABLE simplified program understanding, this lack of prior knowledge about READABLE meant that a positive result could be treated as an even stronger demonstration of READABLE's effectiveness.

The subject pool included 81 students taking different programming courses at a US public university. To elicit participation, subjects were informed that the experiment would help them evaluate their own ability to understand program code written by an external party. No external reward was provided to subjects for participation/non-participation. Of the 81 subjects, 42 elected to participate for a response rate of 52%. To minimize extraneous effects on the experiment, subjects were randomized to the treatment and control groups. Out of a total of 40 subjects assigned to the treatment group, 20 respondents elected to participate. Of 41 subjects assigned to the control group, 22 elected to participate. Thus, the experiment satisfied the recommended minimum sample size constraint of 20 observations per group [33].

4.2. Results

Tests were performed for the successful randomization of subjects, and the impact of non-responses on the experimental design. To test for successful randomization, a post-hoc analysis was conducted to compare demographic characteristics of the subjects. Differences between age, years of work experience, years of IT experience, years of programming experience, and likelihood of participation given assignment to either the treatment or control group were evaluated using two way independent sample *t*-tests. Possible differences in groups due to gender and educational experience were tested using a chi-squared test. No significant differences at the $\alpha \leq 0.05$ level were found between the groups. Table 6 presents the test results.

To test for non-response bias, two-way one sample *t*-tests were conducted to compare characteristics of the respondent sample against known characteristics of the population of information systems students at the university. Tests were performed on gender and work experience. None of the results were significant suggesting that the sample was an accurate representation of the population being studied. The university did not collect statistics for the other demographic variables employed in the study.

One-tailed independent sample *t*-tests were administered to measure the results of hypotheses 1–3. A two-tailed independent sample *t*-test was administered to measure the result of hypothesis 4, as no direction was hypothesized. All tests were performed at the $\alpha \leq 0.05$ level of significance and $1-\beta \geq 0.8$ level of

Table 6
Differences between demographic groups

(a) *T*-test

Demographic	Treatment mean (S.D.)	Control mean (S.D.)	*t*	*N*	*p*-value
Age	26.882 (5.159)	26.571 (6.153)	0.166	38	0.869
Work experience	5.235 (5.093)	4.950 (6.065)	0.153	37	0.879
IT work experience	0.688 (1.137)	0.611 (1.335)	0.183	35	0.856
Programming experience	0.582 (1.050)	0.944 (1.381)	−0.869	35	0.391
Likelihood of participation	0.500 (0.506)	0.537 (0.505)	−0.326	81	0.746

(b) χ^2-test

Demographic	χ^2	*N*	*p*-value
Gender	4.011	42	0.135
Education	0.453	38	0.797

Table 7
Program understanding of READABLE—adjusted scores

	A1: mean (S.D.)	A2: mean (S.D.)	Effect size	t	p-value	95% Conf. interval (min)	95% Conf. interval (max)	Interpretation
H1	0.800 (2.167)	−0.364 (2.013)	0.537	1.804	0.040	0.078	N/A	READABLE control flow is *significantly easier* to understand
H2	0.700 (2.080)	−0.227 (2.022)	0.445	1.464	0.076	−0.139	N/A	READABLE program states may be *significantly easier* to understand
H3	0.250 (1.970)	0.909 (2.389)	0.276	−0.970	0.169	−1.804	N/A	Data Flow is *unlikely to be worse* in READABLE
H4	0.750 (2.173)	0.545 (2.132)	0.094	0.308	0.760	−1.139	1.548	*No differences* can be found between the groups

power. Cohen's d [19], a measure of the effect size (magnitude of difference) was also calculated. Table 7 summarizes the adjusted results (i.e., factoring in guessing). Table 8 summarizes the results for correct answers only.

Despite the small sample size, the results in both Tables 7 and 8 suggest that software developed with the READABLE approach is easier to understand than that developed with traditional Java. For both the raw and adjusted scores, the key hypothesis of interest (H1) was statistically significant, which indicates that respondents were better able to understand control flow in a READABLE program than that written in traditional Java. Although the small sample sizes did not allow us to accept the null hypothesis for H4, the small effect sizes suggest that any contamination was small or practically insignificant. When scores were not adjusted, the effect size was 0.247. When scores were adjusted, the effect size was 0.095. Cohen [19] suggests that an effect size of 0.2 is a "weak effect". Thus, given the small sample size, we conclude that there is sufficient evidence to accept the null hypothesis (H4) that the treatment and control groups did not materially differ on their understanding of the individual modules. It is not possible to ascertain whether READABLE has any adverse impact on respondents' understanding of a program's states (H2) or data flow (H3). However, given the low effect size of any possible adverse impact, we interpret this to mean that for program states and data flow, READABLE did not practically differ from traditional Java. Combined, these tests of hypotheses indicate that maintainers find it easier to follow control flow in READABLE without potential adverse effects on understanding of other contructs such as program states, data flow and individual modules. By implication, READABLE code is thus more easy to maintain than code written in a traditional way.

5. Discussion

We have described a novel approach called READABLE for developing applications (especially multi-paradigm applications such as those for the web). The approach facilitates communication of control flows between the developer and the maintainer, thereby simplifying program understanding. The empirical

Table 8
Program understanding of READABLE—raw scores only

	A1: mean (S.D.)	A2: mean (S.D.)	Effect size	t	p-value	95% Conf. interval (min)	95% Conf. interval (max)	Interpretation
H1	3.100 (1.410)	2.318 (1.359)	0.575	1.829	0.038	0.062	N/A	READABLE control flow is *significantly easier* to understand
H2	3.000 (1.589)	2.318 (1.323)	0.515	1.516	0.069	−0.075	N/A	READABLE program states may be *significantly easier* to understand
H3	2.800 (1.642)	2.818 (1.563)	0.012	−0.037	N/A	N/A	N/A	Data Flow is *unlikely to be worse* in READABLE
H4	3.050 (1.959)	2.591 (1.764)	0.260	0.799	0.214	−0.702	1.620	Likely that *no differences* can be found between the groups

results suggest that READABLE does succeed in improving program understanding for the maintainer, thereby simplifying maintenance. The positive result for hypothesis H1 and the lack of negative results for hypotheses H2 to H4 suggest that READABLE contains the potential to be an effective approach for developing maintainable software.

The experimental READABLE approach represents a nexus of multiple theoretical bases drawn from the research streams of program understanding and software documentation. Specifically, the approach draws on Pennington's [56] constructs and Parnas [53] and Peters and Parnas's [58] suggestion that tabular representations can facilitate program understanding behaviors [82,83] regardless of the maintainers' knowledge of the application domain [62,89]. The key contribution of this research is this conceptual exercise and its operationalization. The implementation we have described in this paper represents one primitive instantiation of this experimental approach for the Java programming language. This primitive implementation nevertheless demonstrates the feasibility and usefulness of our approach. Other, more elegant and sophisticated implementations using state charts or graphical interfaces can be developed. The approach developed in this paper can form the basis of further work of this nature. More efficient organization and retrieval mechanisms can be developed as well. This paper's goal was to demonstrate the feasibility and establish potential usefulness of the READABLE approach; these extensions are, therefore, beyond the scope of the current study.

5.1. Benefits

As a software artifact, the READABLE execution environment is a prototype, and hence cannot be deployed directly for commercial use. Nevertheless, it is useful to conjecture about the role of the READABLE approach in supporting commercial application development. We suggest that READABLE will be most suited to developing new applications instead of retrofitting existing ones. Retrofitting is likely to be expensive because control flow and other programming information from disparate sources must be collated. Arguably, such an exercise would have additional benefits. For example, the exercise would help the maintainer understand an application's control flow. Nevertheless, for many applications, the up front costs of this exercise may overshadow potential future benefits. For new application development, however, developing a program in the manner reported in this paper will likely present few additional demands on the programmer with the potential of easier maintenance.

With increasing acceptance of the web as the primary programming and software deployment platform, software evolution and maintainability concerns will continue to be important. Another contributing factor to this problem is the increasing complexity of web applications as diverse technologies co-exist in the web application development space. One recent addition to this mix of technologies is the web services platform. This set of standards suggests a clear separation between Web Services Description Language (WSDL) for specification of the services, and Business Process Execution Language for Web Services (BPEL4WS). The READABLE approach would, therefore, augment the creation and maintenance of processes that represent the composition of web services by providing an explicit specification of the control layer. A second set of technologies, which underlie the web services platform is XML (eXtensible Markup Language) and XSL (eXtensible Stylesheet Language), which is being used to replace the HTML presentation component in older web applications. The READABLE approach can accommodate the XML/XSL combination in place of HTML by simply leveraging the requisite functionality in XSL to identify and route the module to the appropriate processor. The mode of execution of a READABLE application would not change.

The READABLE approach can also be used to implement the Model-View-Controller (MVC) architecture [30] (among others). The ideas underlying the MVC architecture suggest a separation of concerns between models (underlying schemas), views (visible to different users), and control (manner of use by different users). The READABLE approach builds on the idea of separation of concerns to provide the maintainer easy to manipulate representations of the MVC controller layer. This also serves as the communication between the developer and the maintainer. The example presented in this paper, in fact, employs the MVC architecture, where a user inputs some information in a form. The benefits of READABLE are, therefore, more directly visible within the context of applications deployed with the MVC architecture.

However, the READABLE approach provides additional benefits such as the ability to handle heterogeneous environments, and integration of numerous technologies (e.g., web services, HTML forms, XML data) and is not limited to applications developed with the MVC architecture. The *N*-Queens problem employed in our empirical test, for example, does not closely follow the MVC application. The tables in a READABLE implementation (e.g., control-flow) replace the controllers found in a Model 2 MVC.

The use of READABLE code to illustrate the MVC architecture has particular value in pedagogy. In this paper, we have focused on READABLE as a new way of developing software. However, READABLE can also be employed as a way to illustrate the use of tables in software design. As mentioned earlier, Parnas and colleagues have demonstrated that tables can be useful for specifying code before actual implementation [53,58]. The READABLE approach provides a way for novices to visualize how the table specification might be implemented in actual code.

5.2. Study limitations

The generalizability of our findings is limited by the scope of the empirical study such as the small sample size, and use of students as subjects. Resolving these issues is beyond the scope of the present study. Further studies are required to empirically test usefulness of the approach for multiple tasks of different sizes and for expert programmers. Pragmatic constraints on resources prevent us from performing these additional experiments or more robust implementations that may allow use of the READABLE approach with other programming languages. We have demonstrated that as an experimental approach, READABLE is feasible, and the limited evaluation we report shows that it can be useful for the purpose of improving program understanding (i.e., for developing maintainable software).

Additional studies may also be performed by allowing subjects to mimic programming practices such as access to CASE tools, integrated development environments, or other technologies to help them understand the code. These would investigate the process developers may follow for authoring programs instead of the focus of our investigation, which was to understand how maintainers would understand a program authored by someone else. It is, however, possible to speculate possible synergies between READABLE and CASE Tools. For example, such tools can be employed to provide a graphical interface to the READABLE metamodel. Our research also suggests a new research approach to software development. In this research approach, information for development and maintenance is not embedded in a tool, but instead is somehow integrated with the source code. In this way, developers and maintainers can make use of the information even when the tool is unavailable.

It is difficult to ascertain the scalability of READABLE because it currently exists as a research prototype. However, there is some evidence to suggest that the READABLE approach and the accompanying environment would be scalable in three ways.

- *Operational Performance:* The elements contained in the READABLE approach are represented in a tabular form, i.e., they can be stored in a relational database. Most database queries would, therefore, require a time of $O(n \log n)$ to execute as they directly query primary keys. Second, the research prototype of READABLE has been used to author and deploy multiple applications of different sizes in varied domains. One of these, the IS Bibliographic Repository [17] contains information on approximately 80 thousand journal article entries.
- *Understanding Large Systems:* Another test of scalability is whether the READABLE approach can facilitate improved program understanding of large-scale systems. Experimental methods generally cannot be applied to such settings. Instead, we present the following argument. With the READABLE approach, the control flow is explicitly documented and available to the maintainer in a central location. As the scale of the application grows, the availability of such documentation should be even more valuable because less time is spent tracking down control flow scattered across modules. Further support to our argument is provided by prior research suggesting that tables are most useful as a representational format when a task is complex [76,95]. Furthermore, as READABLE code is stored in a database, it is possible to leverage operators such as 'Project' and database search mechanisms to find relevant sections of the tabular representations. While

searches for code in traditional application development environments are possible, such searches are often difficult because the unit of storage (the source code file) is not optimized for search. Finally, it may be possible to extend the tabular representations in READABLE to more complex ones such as nested tables that hide complexities across different levels of abstraction to facilitate understanding of large-scale systems.

- *Understanding in Multi-developer Settings:* The READABLE approach can also be adapted to settings where multiple developers work on the same project. In traditional project settings involving large numbers of developers, the file is the smallest unit of information shared. Thus, one developer requiring information is often exposed to code irrelevant to his or her task, but nevertheless captured in the file containing the required information. Because READABLE operates from databases, it is possible to provide developers with code-sharing at a finer level of granularity. One can create views on the READABLE code so that a developer is only exposed to code relevant to the developer's task.

The discussion above is especially relevant for applications that are distributed over *n*-tier architectures that may include a multitude of disparate servers. In such cases, it is likely that a maintainer will not have access to a portion of the code deployed on remote servers. Following a READABLE specification, the maintainer would still be able to determine problem situations and if possible, adjust code that is under his/her control (treating other modules as "black box" modules) to either arrive at desired results or coordinate maintenance efforts with other developers. Empirical investigations of these remain beyond the scope of the current study.

6. Conclusion

The approach we have proposed for developing self-documenting and maintainable software for the web can significantly improve the practice and outcome of software created for this platform. The approach represents program components (e.g., HTML, Java Servlets) and their control-flows as tuples in a relational database. A corresponding environment uses the tuples in the relational database to execute the web application. The feasibility of the approach and environment has been demonstrated by extending Java and implementing multiple prototype applications. A limited empirical assessment of the approach has shown that READABLE facilitates program understanding. The results, thus, demonstrate that READABLE is both viable and useful for creating self-documenting web applications that promote greater program understanding, and hence become more maintainable. Our current research focuses on the application of READABLE towards developing software factories to facilitate the (semi) automatic manufacture of integrated software applications [22]. We are also studying the feasibility of integrating our approach and environment with CASE tools.

The READABLE approach provides opportunities for work in additional research streams. For example, it would be interesting to ascertain whether READABLE could be adapted successfully to other popular programming languages such as Visual Basic. Alternate experiments could be conducted on these programming languages to determine whether READABLE successfully enhances program understanding. The READABLE approach also relies on encapsulation as a mechanism for reuse. Adding other mechanisms such as polymorphism and inheritance to READABLE would further improve robustness of the approach, and increase its applicability to a wide variety of problems.

Acknowledgements

We would like to thank the editor, associate editor, and reviewers for their comments on a previous version of this paper. We also wish to thank Matti Rossi, Alexander Hars, Gayle Beyah, Lan Cao, Karlene Cousins, Kannan Mohan, Han Gyun Woo, Roger Chiang, Detmar Straub, Peng Xu, Nicholas Romano, Gerardo Canfora and the reviewers for AMCIS 2003 for comments on an earlier draft of this paper.

This research was supported by the J. Mack Robinson School of Business, Georgia State University, Nanyang Technological University, and Pennsylvania State University.

Appendix A. Appendix

A.1. Experiment source code

Note from the programmer: This company adopts an unusual practice. The main() module is implemented as a table. An example table for a program that detects whether a whole number is even or odd is presented below:

Initial module	Condition	Next module
Begin	True	Seed=Initialize_Module()
Initialize_module	True	Calculate_Numbers (Seed)
Calculate_Numbers	Numbers <0	Calculate_Numbers (Seed)
Calculate_Numbers	Numbers %2=1	Output_Odd()
Calculate_Numbers	Numbers %2=0	Output_Even()
Output_Odd	True	End
Output_Even	True	End

The program in this table can be described as follows. Start with the Initialize Module. Return the value of Initialize Module to Seed. Next, execute the Calculate_Numbers module with the Seed argument. Return the result to Numbers. If Numbers is less than 0, run Calculate_Numbers again. If Numbers divided by 2 gives a remainder of 1 (i.e. an odd number), output odd. Otherwise, output even. After outputting odd or even, end the program.

Initial module	Condition	Next module
Begin	True	boardsize=askBoardSize ()
AskBoardSize	boardsize=0	boardsize=askBoardSize ()
AskBoardSize	boardsize>0	cur_sol=initsol (boardsize)
Initsol	True	solve_queens (cur_sol,boardsize)
solve_queens	cur_sol [0]>−1	no_sol=no_sol+1
solve_queens	cur_sol [0]=−1 && no_sol>0	X=0
no_sol+1	True	solutions=add_solution(cur_sol, solutions,boardsize,no_sol)
add_solution	cur_sol>−1 || no_sol=0	solve_queens(cur_sol,boardsize)
X=0	X<no_solutions	solution(x)
Solution	True	X=x+1
X+1	X<no_solutions	Solution(x)
X=0	X=no_solutions	End
X+1	X=no_solutions()	End

Source code:

```
import java.io.*;
   int no_sol=0;
   int[ ] solutions=null;
   int boardsize=0;
private int[ ] add_solution (int[ ] cur_sol, int[ ] solutions,
                                int boardsize, int no_sol)
   {
     int[ ] sol_array=new int[boardsize*no_sol];
     int x=0;
     for (x=0;x<boardsize*(no_sol–1);x++)
       sol_array[x]=solutions[x];
     for (x=0;x<boardsize;x++)
       sol_array[boardsize*(no_sol–1)+x]=cur_sol[x];
    return sol_array;
   }
private int[ ] solve_queens (int[ ] tested, int boardsize)
   {
     int curline=0;
     int curlinesol=–1;
     int x=0;
     while (curline<boardsize)
      {
      curlinesol=solve_line (tested, boardsize, curline, tested [curline]);
      if (curlinesol==boardsize)
       {
         tested[curline]=–1;
         curline—;
         if (curline<0)
            return tested;
       }
      else
       {
         tested [curline]=curlinesol;
         curline++;
       }
      }
     return tested;
   }
private int solve_line (int[ ] tested, int boardsize,int line)
   {
     return solve_line (tested, boardsize, line,–1);
   }
private int solve_line (int[ ] tested, int boardsize, int line, int prevused)
   {
   int current=prevused+1;
    if (current==boardsize)
     return current;
    int x=0;
    while (x<line && !(tested[x]==current)
            && !(tested[x]+(line–x)==current)
            && !(tested[x]–(line–x)==current))
     x++;
    if (x<line)
     return solve_line (tested, boardsize, line, prevused+1);
    return current;
```

```
   }
private int askBoardSize ( )
   {
    try {
      Integer I=null;
      BufferedReader in=null;
      in=new BufferedReader (new InputStreamReader(System.in));
      String s=null;
      System.out.print ("What are the dimensions of the square board?:");
      s=in.readLine ( );
      I=new Integer (s);
      return I.intValue ( );
      }
      catch (IOException io)
      {
       System.out.println ("IOException");
       return –1;
      }
   }
private int [ ] initsol (int boardsize)
   {
    int [ ] checksol=new int[boardsize];
    int x=0;
    for (x=0;x<boardsize;x++)
       checksol [x]=–1;
    return checksol;
   }
public int no_solutions ( )
   {
    return no_sol;
   }
public String solution (int sol)
   {
    int x=0;
    int y=0;
    int z=0;
    StringBuffer retString=new StringBuffer ( );
    for (x=0;x<boardsize;x++)
     {
      y=solutions [boardsize*(sol)+x];
      for (z=0;z<y;z++)
        retString.append (".");
      retString.append ("Q");
      for (z=y+1;z<boardsize;z++)
        retString.append (".");
      retString.append ("\n");
     }
    return retString.toString ( );
   }
public String num_sol (int sol, int[ ] tosee)
   {
    int x=0;
    StringBuffer retString=new StringBuffer ( );
    for (x=0;x<boardsize;x++)
      retString.append ("Solution"+sol+" line no: "+tosee [x]+"\n");
    return retString.toString ( );
   }
```

Appendix B. Demographic Questionnaire

Gender: Male Female
Current educational level: Freshman Sophomore Junior Senior Masters PhD
Please fill in the blanks
Age: _____
Major: _____
No. years work experience: _____
Current job: _____
No. of years of IT work experience: _____
No. of years of programming experience: _____
Programming languages known:
_____
_____
_____
_____
_____
_____

A.1. Program Understanding Questionnaire

For each question, circle Y if the answer to the question is Yes. Circle N if the answer is No. Both variables and functions are identified by the Arial font. Functions are distinguished using parentheses (). Array indexes are distinguished using square brackets [].

Control flow

Y	N	1.	The minimum number of times solve_queens() will be called is no_sol+1.
Y	N	5.	add_solution() can be called after solution().
Y	N	9.	solve_queens() can be called after add_solution().
Y	N	13.	initsol() can be called more than once.
Y	N	17.	In solve_queens(), when curline is 3, solve_line() has been called at least three times.

Program States

Y	N	3.	The minimum possible boardsize is 0.
Y	N	7.	In solve_line(), it is possible for two values in the tested array to have the same value.
Y	N	11.	add_solution() appends cur_sol to solution.
Y	N	15.	In solution(), the number of '.' Per line is 1 less than boardsize.
Y	N	19.	Every time the program is executed, askBoardSize() will be called.

Data Flow

Y	N	2.	The array returned by solve_queens() becomes cur_sol.
Y	N	6.	boardsize determines the dimensions of cur_sol
Y	N	10.	The integer line in solve_line() is the same as no_sol.
Y	N	14.	In solve_queens(), tested[current] can be greater than boardsize.
Y	N	18.	s in askBoardSize(), is the same as boardsize.

Modules

Y	N	4.	When the solve_queens() function exits, curline is always equal to boardsize.
Y	N	8.	After askBoardSize() is executed, cur_sol is an array whose values are all −1.
Y	N	12.	If boardsize is greater than 0, solve_queens() is called at least once.
Y	N	16.	In solve_queens(), when curline is 3, tested [2] must be less than 3.
Y	N	20.	In solveline(), if tested [3] is 3, tested [1] can be 1.

References

[1] J.R. Anderson, W.D. Gray, Change-episodes in coding: when and how do programmers change their code? Proceedings of the Second Workshop on Empirical Studies of Programmers, 1987.

[2] M. Aoyama, Web-based agile software development, IEEE Software 15 (6) (1998) 56–65.

[3] M.N. Armstrong, C. Trudeau, Evaluating architectural extraction tools, Proceedings of the 5th Working Conference on Reverse Engineering, 1998.

[4] D.L. Atkins, T. Ball, G. Bruns, K.C. Cox, Mawl: a domain-specific language for form-based services, IEEE Transactions on Software Engineering 25 (3) (1999) 334–346.

[5] R.L. Baskerville, L. Levine, J. Pries-Heje, B. Ramesh, S.A. Slaughter, How internet software companies negotiate quality, IEEE Computer 34 (5) (2001) 51–57.

[6] T. Berners-Lee, R. Cailliau, WorldWideWeb: proposal for a hypertext project, www.w3.org/Proposal.html, 1990.

[7] B.W. Boehm, Software engineering, IEEE Transactions on Computer 25 (1976) 1226–1241.

[8] G.H. Bower, J.B. Black, T.J. Turner, Scripts in memory for text, Cognitive Psychology 11 (1979) 177–220.

[9] C. Brabrand, A. Moller, M.I. Schwartzbach, The <bigwig> project, ACM Transactions on Internet Technology 2 (2) (2002) 79–114.

[10] R. Brooks, Towards a theory of the comprehension of computer programs, International Journal of Man–Machine Studies 39 (2) (1983) 237–267.

[11] J.C. Brown, D.B. Johnson, FAST: a second generation program analysis system, Proceedings of the Second International Conference on Software Engineering, IEEE Press, Atlanta, GA, 1978.

[12] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. Grose, Eclipse Modeling Framework, Addison-Wesley Professional, Boston, MA, 2003.

[13] R. Cailliau, H. Ashman, Hypertext in the web—a history, ACM Computing Surveys 31 (4es) (1999) 1–6.

[14] C. Cavaness, Programming Jakarta Struts, O'Reilly and Associates, Cambridge, MA, 2002.

[15] Y.F. Chen, M.Y. Nishimoto, C.V. Ramamoorthy, The C information abstraction system, IEEE Transactions on Software Engineering 16 (3) (1990) 325–334.

[16] R.H.L. Chiang, A knowledge-based system for performing reverse engineering of relational databases, Decision Support Systems 13 (3–4) (1995) 295–312.

[17] C.E.H. Chua, L. Cao, K. Cousins, K. Mohan, D.W. Straub, V.K. Vaishnavi, IS Bibliographic Repository (ISBIB): a central repository of research information for the IS community, Communications of the AIS 8 (27) (2002) 392–412.

[18] C.E.H. Chua, V.C. Storey, R.H.L. Chiang, A software engineering environment for search engine development, Proceedings of the 12th Workshop on Information Technology and Systems, 2002.

[19] J. Cohen, Statistical power analysis for the behavioral sciences, Lawrence Erlbaum Associates, 1988 Hillsdale,NJ.

[20] R.A. Coll, J.H. Coll, G. Thakur, Graphs and tables: a four factor experiment, Communications of the ACM 37 (1994) 77–86.

[21] T.D. Cook, D.T. Campbell, Quasi-experimentation: Design and Analysis Issues for Field Settings, Houghton, Mifflin and Company, Boston, MA, 1979.

[22] M.A. Cusumano, The software factory: a historical interpretation, IEEE Software 6 (2) (1989) 23–30.

[23] S.P. Davies, Models and theories of programming strategy, International Journal of Man–Machine Studies 39 (2) (1993) 237–267.

[24] S. Demeyer, S. Ducasse, S. Tichelaar, Why unified is not universal, Proceedings of the Second International Conference on the Unified Modeling language, 1999.

[25] F. Douglis, S.J. Chapin, J. Isaak, Technical activities forum: internet research on internet time, IEEE Computer 31 (11) (1998) 76–77.

[26] M.F. Fernandez, D. Florescu, J. Kang, A.Y. Levy, D. Suciu, Catching the boat with strudel: experiences with a web-site management system, Proceedings of the SIGMOD Conference on Management of Data, 1998.

[27] A. Forward, T.C. Lethbridge, The relevance of software documentation, tools and technologies: a survey, Proceedings of the 2002 ACM Symposium on Document Engineering, 2002.

[28] R. Fourer, Database structures for mathematical programming models, Decision Support Systems 20 (4) (1997) 317–344.

[29] P. Fraternali, Tools and approaches for developing data-intensive web applications: a survey, ACM Computing Surveys 31 (3) (1999) 227–263.

[30] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: elements of reusable object-oriented software, Addison-Wesley, 1995.

[31] R. Guindon, Knowledge exploited by experts during software system design, International Journal of Man–Machine Studies 33 (3) (1990) 279–304.
[32] J. Gulden, The reverse engineering Java API Documentation Generator, http://classdoc.sourceforge.net/classdoc10/classdoc.html, 2002.
[33] J.F. Hair Jr., R.E. Anderson, R.L. Tatham, W.C. Black, Multivariate Data Analysis with Readings, Fifth edition, Prentice-Hall, Upper Saddle River, NJ, 1998.
[34] D. Harel, State charts: a visual formalism for complex systems, Science of Computer Programming 8 (3) (1987) 231–275.
[35] D. Harel, E. Gery, Executable object modeling with statecharts, Computer 30 (7) (1997) 1–42.
[36] A. Holub, When it comes to good OO design, keep it simple, JavaWorld (2002).
[37] S.C. Johnson, Yacc: Yet Another Compiler Compiler, UNIX Programmer's Manual, vol. 2, Holt, Rinehart, and Winston, New York, NY, USA, 1979, pp. 353–387.
[38] T.C. Jones, Estimation of Software Costs, McGraw/Hill, 1998.
[39] S.O. Kimbrough, Y. Yang, Action at the tables: sketching a tabular representation for utterances under the language–action perspective, Proceedings of the Language–action Perspective, 2004, New Brunswick, NJ.
[40] D.E. Knuth, Literate Programming, Center for the Study of Language and Information, Stanford, CA, 1992.
[41] J. Kotula, Source code documentation: an engineering deliverable, Proceedings of the Technology of Object-oriented Languages and Systems, 2000.
[42] B.P. Lientz, E.B. Swanson, Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations, Addison-Wesley, Reading, MA, 1980.
[43] B.P. Lientz, E.B. Swanson, Problems in application software maintenance, Communications of the ACM 24 (11) (1981) 763–769.
[44] B.P. Lientz, E.B. Swanson, G.E. Tompkins, Characteristics of application software maintenance, Communications of the ACM 21 (6) (1978) 466–471.
[45] M.A. Linton, Implementing relational views of programs, Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 1984.
[46] J.R. Marsden, R. Pakath, K. Wibowo, Decision making under time pressure with different information sources and performance-based financial incentives—Part 1, Decision Support Systems 34 (1) (2002) 75–97.
[47] J.C. Martin, Introduction to Languages and the Theory of Computation, WCB/McGraw-Hill Companies, 1997.
[48] R.E. Mayer, The psychology of how novices learn computer programming, in: E. Soloway, J.C. Spohrer (Eds.), Studying the Novice Programmer, Lawrence Erlbaum Associates, 1989, pp. 129–159.
[49] G. Mecca, P. Atzeni, A. Masci, G. Sindoni, P. Merialdo, The Araneus web-based management system, Proceedings of the ACM SIGMOD International Conference on Management of Data, 1998.
[50] K. Nørmark, Requirements for an elucidative programming environment, Proceedings of the 8th International Workshop on Program Comprehension, 2000.
[51] T. Olsson, J. Grundy, Supporting traceability and inconsistency management between software artifacts, Proceedings of the IASTED International Conference on Software Engineering and applications, 2002.
[52] J. Paakki, A. Salminen, J. Koskinen, Automated hypertext support for software maintenance, The Computer Journal 39 (7) (1996) 577–599.
[53] D.L. Parnas, J. Madey, M. Iglewski, Precise documentation of well-structured programs, IEEE Transactions on Software Engineering 20 (12) (1994) 948–976.
[54] M.C. Paulk, Extreme programming from a CMM perspective, IEEE Software 18 (6) (2001) 19–26.
[55] A. Pellegrin, M. Bétrancourt, How can spatial arrangement in tables improve readers' cognitive processing? Proceedings of the AAAI 2000 Spring Symposium Series: "Smart Graphics", 2000, Stanford, CA.
[56] N. Pennington, Stimulus structures and mental representation in expert comprehension of computer programs, Cognitive Psychology 19 (1987) 295–341.
[57] D.N. Perkins, C. Hancock, R. Hobbs, F. Martin, R. Simmons, Conditions of learning in novice programmers, in: E. Soloway, J.C. Spohrer (Eds.), Studying the Novice Programmer, Lawrence Erlbaum Associates, 1989, pp. 261–279.
[58] D.K. Peters, D.L. Parnas, Using test oracles generated from program documentation, IEEE Transactions on Software Engineering 24 (3) (1998) 161–173.
[59] M. Pizka, STA—a conceptual model for system evolution, Proceedings of the International Conference on Software Maintenance, IEEE Computer Society, Montreal, Canada, 2002.
[60] T. Quatrani, Visual Modeling with Rational Rose 2002 and UML, Pearson Education, 2002.
[61] V. Rajlich, J. Doran, R. Gudla, Layered explanations of software: a methodology for program comprehension, Proceedings of the Third Workshop on Program Comprehension, 1994.
[62] J.E. Robbins, D.F. Redmiles, Software architecture design from the perspective of human cognitive needs, Proceedings of the California Software Symposium, 1996.
[63] D.E. Rumelhart, Schemata: the building blocks of cognition, in: R.J. Spiro, B.C. Bruce, W.F. Brewer (Eds.), Theoretical Issues in Reading Comprehension, Lawrence Erlbaum Associates, 1980, pp. 33–58.
[64] J. Sametinger, M. Riebisch, Evolution support by homogeneously documenting patterns, aspects and traces, Proceedings of the 6th European Conference on Software Maintenance and Reengineering, 2002.
[65] D. Schwabe, R.M. Guimarães, G. Rossi, Cohesive design of personalized web applications, IEEE Internet Computing 6 (2) (2002) 34–43.
[66] D. Schwabe, G. Rossi, An object oriented approach to web-based applications design, Theory and Practice of Object Systems 4 (4) (1998) 207–225.
[67] D. Sharon, Meeting the challenge of software maintenance, IEEE Software 13 (1) (1996) 122–125.

[68] J.P. Shim, M. Warkentin, J.F. Courtney, D.J. Power, R. Sharda, C. Carlsson, Past, present, and future of decision support technology, Decision Support Systems 33 (2) (2002) 111–126.
[69] J.B. Smelcer, E. Carmel, The effectiveness of different representations for managerial problem solving: comparing tables and maps, Decision Sciences 28 (2) (1997) 391–420.
[70] K. Smolander, K. Lyytinen, V.P. Tahvanainen, P. Marttiin, MetaEdit—a flexible graphical environment for methodology modelling, Proceedings of the Advanced Information Systems Eengineering (CAiSE) conference, 1991, Trondheim, Norway.
[71] E. Soloway, K. Ehrlich, Empirical studies of programming knowledge, ACM Transactions on Software Engineering 10 (5) (1984) 595–609.
[72] J.L. Steffen, Interactive examination of a C program with CScope, Proceedings of the USENIX Association Winter Conference, 1985.
[73] M.-A.D. Storey, K. Wong, H.A. Müller, How do program understanding tools affect how programmers understand programs? Proceedings of the Fourth Working Conference on Reverse Engineering, 1997.
[74] Sun microsystems, Javadoc Tool Home Page, 2002.
[75] Y. Takahara, N. Shiba, H. Tanaka, An implementation of unified programming on actDSS, Decision Support Systems 18 (3–4) (1996) 273–292.
[76] D. Te'eni, Determinants and consequences of perceived complexity in human–computer interaction, Decision Sciences 20 (1) (1989) 166–181.
[77] S.R. Tilley, H.A. Müller, M.A. Orgun, Documenting software systems with views, Proceedings of the 10th International Conference on Systems Documentation, 1992.
[78] A.M. Turing, On computable numbers with an application to the Entscheidungsproblem, Proceedings of the London Mathematical Society 2 (42) (1936) 230–265.
[79] A.M. Turing, Computing machinery and intelligence, Mind 59 (1950) 433–560.
[80] N.S. Umanath, I. Vessey, Multiattribute data presentation and human judgement: a cognitive fit perspective, Decision Sciences 25 (5/6) (1994) 795–824.
[81] A. van Deursen, T. Kuipers, Building documentation generators, Proceedings of the International Conference on Software Maintenance, 1999.
[82] A.M. Vans, A.A.v. Mayrhauser, G. Solmo, Program understanding behavior during corrective maintenance of large-scale software, International Journal of Human Computer Studies 51 (1) (1999) 31–70.
[83] I. Vessey, Expertise in debugging computer programs: a process analysis, International Journal of Man–Machine Studies 23 (5) (1985) 459–494.
[84] I. Vessey, On matching programmers chunks with program structures: an empirical investigation, International Journal of Man–Machine Studies 27 (1) (1987) 65–89.
[85] I. Vessey, Cognitive fit: a theory-based analysis of the graphs versus tables literature, Decision Sciences 22 (2) (1991) 219–240.
[86] I. Vessey, R. Weber, Some factors affecting program repair maintenance: an empirical study, Communications of the ACM 23 (2) (1983) 128–134.
[87] A.O. Villeneuve, J. Fedorowicz, Understanding expertise in information systems design, or, what's all the fuss about objects? Decision Support Systems 21 (2) (1997) 111–131.
[88] G. Visaggio, Relationships between documentation and maintenance activities, Proceedings of the 5th International Workshop on Program Comprehension, 1997.
[89] A.A. von Mayrhauser, A.M. Vans, From code understanding needs to reverse engineering tool capabilities, Proceedings of the Ssixth International Workshop on Ccomputer-aided Software Engineering, 1993.
[90] A.A. von Mayrhauser, A.M. Vans, Identification of dynamic comprehension processes during large scale maintenance, IEEE Transactions on Software Engineering 22 (6) (1996) 424–437.
[91] R. Waters, S. Rugaber, G.D. Abowd, Architectural element matching using concept analysis, Proceedings of the 14th IEEE International Conference on Automated Software Engineering, 1999.
[92] D.M. Weiss, V.R. Basili, Evaluating software development by analysis of changes: some data from the software engineering laboratory, IEEE Transactions on Software Engineering 11 (1) (1985) 157–168.
[93] K. Wong, S.R. Tilley, H.A. Müller, M.-A.D. Storey, Structural redocumentation: a case study, IEEE Software 12 (1) (1995) 46–54.
[94] S.S. Yau, J.S. Collofello, Design stability measures for software maintenance, IEEE Transactions on Software Engineering 11 (9) (1985) 849–856.
[95] R.W. Zmud, E. Blocher, R.P. Moffie, The impact of color graphic report formats on decision performance and learning, Proceedings of the Fourth International Conference on Information Systems, 1983, Chicago, IL.

Cecil Eng Huang Chua is an assistant professor at Nanyang Technological University. His research interests include (1) reconstructing the relationship between database and software engineering theories, (2) understanding the role of the IS researcher, and (3) misappropriation of technology. Cecil has several publications in such journals as *Data and Knowledge Engineering*, *Decision Support Systems*, *Journal of the AIS* and the *VLDB Journal*. Cecil maintains the IS Bibliographic Repository, a store of bibliographic information on IS journals.

Sandeep Purao is associate professor of Information Sciences and Technology at Pennsylvania State University, University Park. His research focuses on information systems in organizations with a particular emphasis on information systems design and workflow management. His work has been published in several journals including *Information Systems Research*, *Communications of the ACM*, *Information & Organizations*, *Journal of MIS*, *ACM Computing Surveys*, and *IEEE Transactions on Systems, Man and Cybernetics*.

Veda C. Storey is Tull Professor of Computer Information Systems, J. Mack Robinson College of Business Administration Georgia State University. She has research interests in database management systems, intelligent systems, the Semantic Web, and ontology development. She has served on the editorial board of several journals including *Information Systems Research*, *Management Information Systems Quarterly* and *Data and Knowledge Engineering*.