December 2003

# READABLE: An Approach and an Environment for Developing Maintainable Web Software

Cecil Chua
*Georgia State University*

Sandeep Purao
*Penn State University*

Veda Storey
*Georgia State University*

# READABLE: AN APPROACH AND AN ENVIRONMENT FOR DEVELOPING MAINTAINABLE WEB SOFTWARE

**Cecil Eng Huang Chua**
Georgia State University
**cchua@cis.gsu.edu**

**Sandeep Purao**
Penn State University
**spurao@ist.psu.edu**

**Veda C. Storey**
Georgia State University
**vstorey@gsu.edu**

## Abstract

*Software maintenance is expensive and difficult because software is complex and maintenance requires understanding code written by someone else. A key component of maintainability is program understanding. Program understanding, however, is problematic for software developed on the web because web applications comprise a mix of technologies and because the rapid pace of web development often means formal development practices, including documentation are ignored. This research proposes an approach, called READABLE, that is aimed at improving understanding and maintainability of web applications. The READABLE approach achieves this goal by making the control flow among web application components visible in a separate layer. This layer allows easier program understanding for the software developers, and can also be used by a controller for execution of the application, making the web application self-documenting. A controlled laboratory experiment shows that the READABLE approach improves program understanding.*

**Keywords:** Web application development, maintenance, software engineering, programming architecture, documentation, robust software

## Introduction

Maintenance remains the most expensive aspect of information system development, consuming 50% to 80% of resources (Boehm, 1976). A major contributor to maintenance expense is the non-correspondence between a system specification and code (Visconti and Cook, 1993). The problem is especially prevalent in web applications for two reasons. First, web applications are a hybrid of hyperlinked web pages and programming code (Berners-Lee and Cailliau, 1990; Cailliau and Ashman, 1999; Fraternali, 1999). As a result, traditional software development paradigms are inappropriate. Second, web applications typically have rapid development cycles (Aoyama, 1998; Baskerville, et al., 2001), which make use of formal documentation processes difficult (Forward and Lethbridge, 2002; Visaggio, 1997).

The objective of this research is to create an approach for developing maintainable web software called READABLE, i.e. the Readable, Executable, and Augmentable Database-Linked Environment. Web applications developed using the READABLE approach self-document critical program understanding information such as control flow and are thus easier to understand. The approach is operationalized as an environment that extends the Java programming language.

## Related Research

Three areas contribute to our research – program understanding, software documentation and web application development.

### *Program Understanding*

Research on program understanding relates notions of human psychology and cognition to understand program code. Four constructs are important for understanding programs, (1) function, (2) control flow, (3) data flow, and (4) program states (Pennington, 1987). 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 developers and maintainers different approaches to understanding programs (von Mayrhauser and Vans, 1993) depending on their knowledge of the application domain (Robbins and Redmiles, 1996). Those unfamiliar with the domain first attempt to understand its control flow (Pennington, 1987), which is then used as a framework for understanding the program. Those familiar with the domain search for code or variables that represent required features (Soloway and Ehrlich, 1984). The code or variables are then used as a beacon (Brooks, 1983; Rajlich, et al., 1994). *Beacons* are recognizable landmarks in code that developers and maintainers use to navigate other, unfamiliar, parts of the source code. In both cases, the control flow construct represents the most important piece of information for program understanding (Storey, et al., 1997).

### *Software Documentation*

Software documentation research applies program understanding principles while simultaneously considering the pragmatics of programming and documentation practice. For example, it recognizes that most programmers fail to provide useful documentation of their code, or maintain existing documentation (Forward and Lethbridge, 2002; Visaggio, 1997). Thus, various methods of auto-generating code from documentation or vice-versa have been proposed such as application generators (Johnson, 1979; Quatrani, 2002), literate programming (Knuth, 1992; Nørmark, 2000), and reverse engineering (Chiang, 1995; Sun Microsystems, 2002; van Deursen and Kuipers, 1999). One interesting substream focuses on the representation of programs as tables and demonstrates that such programs are easier to understand (Parnas, et al., 1994; Peters and Parnas, 1998).

### *Software Documentation in Web Applications*

The nature of web application development tends to exacerbate program understanding/software documentation issues. First, web applications are harder to understand, because web applications comprise multiple technologies (e.g. HTML Forms + JSP + Servlets + Beans or HTML Forms + ASP + ActiveX + COM Objects). Second, there is less incentive to document web applications, because of the short cycle time of web application development (Aoyama, 1998; Baskerville, et al., 2001; Douglis, et al., 1998).

Various attempts have been made to solve the multiple paradigm and web development cycle problems. For example, languages and frameworks such as MAWL (Atkins, et al., 1999), <bigwig>(Brabrand, et al., 2002), and Struts (Cavaness, 2002) have been proposed that provide a single framework for web development. These languages and frameworks bring web application development under a single banner. However, these languages and frameworks closely follow the design philosophies of traditional programming languages. Thus, they do not provide mechanisms for aligning system documentation with the source code and remain difficult to read.

## READABLE

This section proposes the READABLE approach for documenting web applications. The goal of the approach is to develop applications with the following characteristics. First, the resulting software is *Readable*. Information that maintainers need to understand the program is made conveniently available. Second, the same software is also *Executable*. The distinction between code and documentation is eliminated. Third, the software is *Augmentable*, that is, real world constraints are incorporated by ensuring that additions and enhancements to the software can be made without affecting the existing users. These characteristics are recognized by developing the application in a *DAtaBase-Linked Environment*. Thus, unlike traditional web application development, core concepts required in program understanding are explicitly surfaced in the READABLE database and not buried in the source code. In other words, in the READABLE style, a programmer/maintainer changes the application by modifying records in a database. The database is both a source of maintenance documentation, and is simultaneously executable programming code. Figure 1 presents the architecture, which demonstrates how the database connects the various modules (components) of a web application.

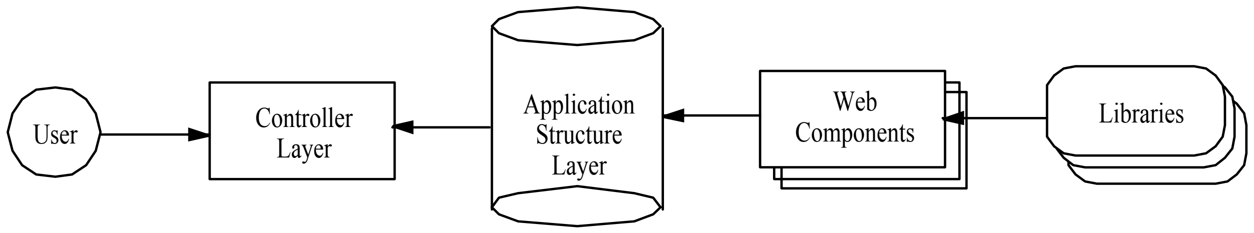

**Figure 1. Structure of a READABLE Application**

The READABLE approach and the accompanying environment consists of four constructs, (1) the control flow, (2) components, (3) variables, and (4) program states. The control flow construct captures the sequence of execution of the submodules. The component construct describes a set of instructions that is executed when a particular event occurs. The variables construct represents a portion of the memory that is either part of the final product, or necessary for developing the final product. The program states construct preserves the memory state after a module has been executed. Figure 2 presents the relationship between the four constructs.

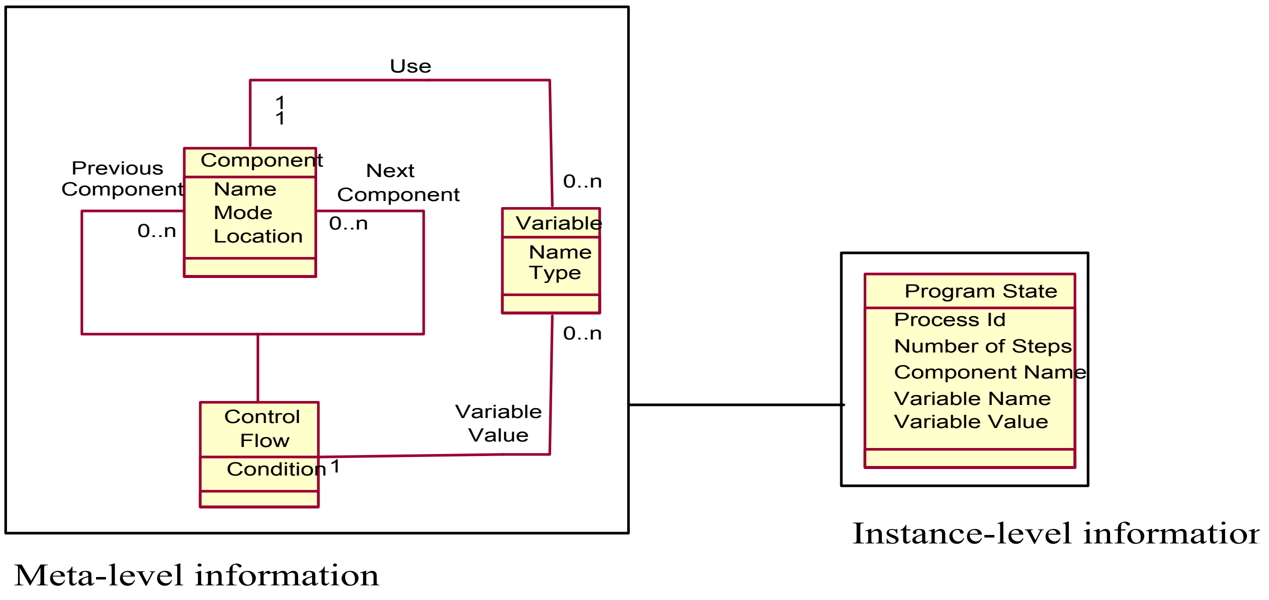

**Figure 2. UML Schema of Application Structure Layer**

To illustrate the operationalization of these constructs, we use a well-known application, the number guessing game that contains all traditional programmatic structure elements such as sequence, iteration and selection, and also requires user (web) interaction.[1] The example therefore demonstrates that any web application can be developed using the READABLE approach. An implementation in READABLE can be found at http://readable.eci.gsu.edu:8080/examples/servlet/demo. In the number guessing game, the user must 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'. The game has the following components: (1) **Genrand** to generate a random

[1]This game is well-known and employed as an example for many other prototypes (e.g., MAWL (Atkins, et al., 1999).

number, (2) **DisplayGuess** to display a screen prompting for a number from the user, (3) **IncNoGuess** to increment the number of guesses attempted, (4) **GoodGuess** to tell the user the guess was correct, (5) **LowGuess** to tell the user that the guess was low, and (6) **HighGuess** to tell the user that the guess was high.

## *Control Flow*

The control flow construct captures the application's possible control flows. It is defined as a 3-tuple, ModuleFlow={Module$_i$, Module$_j$, Condition}. Module$_i$ is referred to as the **Initial Module**, and Module$_j$ as the **Next Module**. This 3-tuple describes all possible control flow configurations of the submodules, including sequence, selection and iteration. The sequence of two modules is captured by specifying the first module as the **Initial Module**, and the second as the **Next Module**. Selection is captured by specifying the **Condition** under which a sequence is allowable. Iteration occurs when the **Next Module** is the **Initial Module** of some previously executed sequence.

Table 1 demonstrates the control flow for the number guessing game. 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, the application should so inform the user (LowGuess). Similarly, if the guess was higher, the application should inform the user (HighGuess). 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. In this manner, conditionals and looping are specified explicitly outside the web components, making them easier to identify, manage and change than in 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.

**Table 1. Example Control Flow Table**

| Record | Initial 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 |

In Figure 1, we presented a controller layer that uses the application structure layer (i.e., the database) to execute the application. The controller layer leverages heavily on the control-flow table to function. The controller layer first searches the control-flow table for the reserved component 'BEGIN' and identifies the one component that transitions from 'BEGIN'. It calls this component the 'current' component. The controller layer then identifies all of the components that the current component can transition to and the conditions that will allow the transitions to take place. The current component is then invoked, with any required variables being passed to it. Execution of the current component changes the controller's state. The controller compares its new state information with the transition conditions. If a transition condition matches its current state, the controller loads the transition component as its new current component, and repeats the cycle. Otherwise, the controller halts.

## *Components*

Components in a READABLE application can be in any language, and may be static web pages, server pages, pages with client scripts, servlets or any other kind of component. The READABLE controller identifies the location of the relevant file, and instructs the appropriate interpreter to execute the component as needed. If a component requires human intervention, the controller halts and alerts the user. The controller retains the following information about the components:

- $\text{Component}_i$, $\text{Component}_j \in$ Components
- $\text{Component}_i$.name = name of component $i \in$ {BEGIN, END or string}
- $\text{Component}_i$.mode = mode of component $i$ Î {0=waiting for user, 1=otherwise}
- $\text{Component}_i$.location = location of component $i$ specified as URI

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.

**Table 2. Example Components Table**

| Name | Mode | Location |
|---|---|---|
| DisplayGuess | Yes | <servername>/<directory>/demo_displayguess |
| GenRand | No | <servername>/<directory>/demo_genrand |
| GoodGuess | Yes | <servername>/<directory>/demo_goodguess |
| HighGuess | No | <servername>/<directory>/demo_guesshigh |
| IncNoGuess | No | <servername>/<directory>/demo_incnoguess |
| LowGuess | No | <servername>/<directory>/demo_guesslow |

## *Variables*

The variables construct captures the existence and current values of variables. This construct records the use of a variable by a component. Upon receiving the request to activate a component, this table is traversed by the controller to ensure that the required variables are instantiated and available. The variables construct and the relationship with the components construct is represented by:

- $\text{Variable}_k \in$ Variables
- $\text{Variable}_k$.name = name of variable $k$
- $\text{Variable}_k$.type = data type of variable $k$
- $\text{Use}_{ik}$ = component $i$ uses variable $k$

**Table 3. Example Variables**

| Component Name | Variable Name | Data Type |
|---|---|---|
| DisplayGuess | Limit | Integer |
| DisplayGuess | NoGuess | Integer |
| DisplayGuess | Guess | Integer |
| DisplayGuess | GuessStatus | String |
| IncNoGuess | NoGuess | Integer |

The example contained in Table 3 demonstrates the variables for the components DisplayGuess and IncNoGuess. These variables are: (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. IncNoGuess increments the NoGuess variable.

## *Program States*

The program states construct represents a concatenation of important elements from the other constructs. It captures the current state and transitions across states for an application instance. In effect, it traces the activity of an application from the time it begins execution to the time it terminates. The program states construct may be understood as a 5-tuple that concatenates specific instantiations from the other constructs for each unique application instance, i.e. for each unique process. This tuple is represented as:

Program State ={ProcessID, NumSteps, ComponentName, VariableName, Value}.

- **Process Id:** A unique value that identifies the current process.
- **NumSteps:** The number of state transitions thus far in the process.
- **Component$_i$.Name:** The name of the component being executed.
- **Variable$_k$.Name:** The name of the variable being logged.
- **Value:** The value of the variable before execution.

## Validation

The feasibility of READABLE has been demonstrated by the implementation of a language based on the style. In addition, an empirical study suggests that the READABLE style promotes program understanding.

### *Feasibility*

We are adapting the Java programming language to the READABLE style. Sufficient features have been incorporated so that prototype applications for other research projects have been developed using the Java/READABLE language. These prototypes demonstrate the feasibility and effectiveness of the style.

- **The IS Bibliographic Repository (Chua, et al., 2002a):** This application stores bibliographic information about journals relevant to information systems research. Unlike other bibliography projects, the IS Bibliography project focuses exclusively on IS research publications and can be found at **http://readable.eci.gsu.edu:8080/examples/servlet/isbib**.

- **The Automated Software Development Environment for Information Retrieval (Chua, et al., 2002b):** 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**.

- **The Entity (Storey, et al., 1998) and Relationship (Ullrich, et al., 2001) Classification Programs:** These programs support the creation of databases and can be found at **http://readable.eci.gsu.edu:8080/examples/servlet/vs_ont** and **http://readable.eci.gsu.edu:8080/examples/servlet/vp**.

### *Ease of Program Understanding*

In the earlier sections, we identified four programming constructs that were important to program understanding (Pennington, 1987): (1) control flow, (2) components, (3) variables, and (4) program states. To improve program understanding, it was imperative that the READABLE approach improves understanding of the more critical constructs (e.g., control flow) while not compromising the less important ones (e.g., components). We therefore designed an experiment to test READABLE's ability to improve program understanding. Four hypotheses were formulated based on the program understanding constructs of Pennington (1987). Table 4 relates the hypotheses to the READABLE constructs. These hypotheses are:

- *H1 (Control Flow):* Subjects would find it easier to understand READABLE's control flow than control flow written using a standard programming language, especially given that READABLE's control flow is explicitly presented in a database table.

- *H2 (Program States):* Subjects would find it easier to understand READABLE's program states than that of a standard programming language.

- *H3 (Data Flow):* Subjects would find it more difficult to understand READABLE's data flow, than the data flow of a standard programming language. Given sufficient statistical power, a failure to reject the null version of this hypothesis (i.e. a failure to confirm this hypothesis) would suggest that the READABLE style did not produce unnecessary effects on other aspects of program understanding.

- *H4 (Components):* Subjects' understanding of READABLE components and components written in the standard programming language would be different. Given sufficient statistical power, a failure to reject the null version of this hypothesis would suggest that extraneous factors did not overly impact the experiment.

Of these four hypotheses, only the first two were of special interest, as they tested for hypothesized benefits of READABLE. The latter two hypotheses were included in the design to test for possible side effects of READABLE (H3) and extraneous influences (H4). 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 the null hypothesis could be accepted.

**Table 4. Relationship Between Hypotheses and READABLE Constructs**

| Hypotheses | READABLE Construct | Traditional Web Development Construct |
|---|---|---|
| H1: Control Flow | Control flow table | Embedded in hyperlinks, <Form Action> commands, procedure calls, etc. |
| H2: Program States | Derived from various READABLE constructs | Derived from source code. |
| H3: Data flow | Derived from parameter passing information | Derived from parameter passing information |
| H4: Components | Derived from component names and component code | Derived from component names and component code |

### *Sample Characteristics*

The study was patterned after the methodology of Pennington (1987) and was conducted on 81 student subjects who took a programming course (5 total classes) from the information systems department of a US public university. Subjects had either taken at least one previous programming course, or were completing their first course and were not taking a course taught by the researchers. Subjects were asked to understand one of two program listings that differed in only one way: Control flow code was represented either in the READABLE style (treatment group), or the traditional object-oriented style (control group). 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 (Hair Jr., et al., 1998).

## *Experimental Procedure*

To minimize the effect of diffusion of treatments, compensatory rivalry, compensatory equalization, and resentful demoralization (Cook and Campbell, 1979), the survey packets administered to the groups were designed to be almost identical, differing only in the specific control (i.e. object-oriented code), or treatment (READABLE code) administered.

The subjects were presented with a scenario where they were hired to maintain code while the main programmer was on vacation. Source code of a Java program that solved the *N*-Queens problem was then presented. The *N*-Queens problem 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. A classic mathematical problem was selected, because of its clean control flow characteristics.

A demographic questionnaire was also administered to test both for equality between groups on extraneous factors, and to test for non-response bias. The survey ended with a 20-item factual yes/no questionnaire that evaluated subjects' understanding of the program listing. Examples of these 20 items and the constructs they mapped to are:

- *H1: Control Flow*: 'add_solution() can be called after solution().' Items 1,5,9,13, and 17 were control flow items.

- *H2: Program States* 'every time the program is executed, askBoardSize will be called.' Items 3,7,11,15, and 19 were program state items.
- *H3: Data Flow* 'the array returned by solve_queens() becomes cur_sol.' Items 2,6,10,14,and 18 were data flow items.
- *H4: Modules* 'when the solve_queens() function exits, curline is always equal to boardsize.' Items 4,8,12,16, and 20 were module items.

The questionnaires were administered under examination conditions. This procedure 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 (Cook and Campbell, 1979). 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. These adjusted scores, and the raw scores (i.e., sum of all correct answers) were used in the analysis.

A learning bias towards traditional programming practice could not be controlled for because the study could only be administered on subjects having some programming knowledge. All subjects knew the traditional programming language, but not READABLE. As the experiment was designed to test whether READABLE simplified program understanding, this meant that a positive result could be treated as a stronger demonstration of READABLE's effectiveness.

## *Analysis and Results*

Statistical analyses were performed both to test for the hypotheses, and possible extraneous effects that could not be controlled for by the experimental design. Specifically, tests were performed for the successful randomization of subjects, and the impact of non-responses on the experimental design. All of these tests were not significant, which suggests that subjects were randomized and that non-response bias was not a material influence.

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 power. Cohen's *d* (Cohen, 1988), a measure of the effect size (magnitude of difference) was also measured. Table 5 summarizes the adjusted results, while Table 6 summarizes the results for correct (raw) answers only.

**Table 5. Program Understanding of READABLE Applications—Adjusted Scores**

| H | Readable Mean (Std. Dev) | Std. Java Mean (Std. Dev) | Effect Size | t-score | p-value | Interpretation |
|---|---|---|---|---|---|---|
| H1 | 0.800 (2.167) | -0.364 (2.013) | 0.558 | 1.804 | 0.040 | READABLE control flow is significantly easier to understand |
| H2 | 0.700 (2.080) | -0.227 (2.022) | 0.452 | 1.464 | 0.076 | READABLE program states may be significantly easier to understand |
| H3 | 0.250 (1.970) | 0.909 (2.389) | 0.301 | -0.970 | 0.169 | Data Flow is not significantly worse in READABLE |
| H4 | 0.750 (2.173) | 0.545 (2.132) | 0.095 | 0.308 | 0.760 | Likely that no differences can be found between the groups |

**Table 6. Program Understanding of READABLE Application—Raw Scores Only**

| H | Readable Mean (Std. Dev) | Std. Java Mean (Std. Dev) | Effect Size | t-score | p-value | Interpretation |
|---|---|---|---|---|---|---|
| H1 | 3.100 (1.410) | 2.318 (1.359) | 0.565 | 1.829 | 0.038 | READABLE control flow is significantly easier to understand |
| H2 | 2.800 (1.642) | 2.818 (1.563) | 0.011 | -0.037 | N/A | READABLE program states are not easier to understand |
| H3 | 3.000 (1.589) | 2.318 (1.323) | 0.470 | 1.516 | 0.069 | Data Flow is not significantly worse in READABLE |
| H4 | 3.050 (1.959) | 2.591 (1.764) | 0.247 | 0.799 | 0.214 | Likely that no differences can be found between the groups |

Despite the small sample size, the results in both Table 5 and Table 6 suggest that the READABLE style is easier to understand than standard Java. For both the raw and adjusted scores, Hypothesis 1 was statistically significant, which indicates that respondents were better able to understand READABLE control flow than standard Java control flow. While the small sample sizes did not allow us to accept the null hypothesis for H4, the small effect sizes do 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 (1988) 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 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 does or does not have any adverse impact on the respondents' understanding of a program's states or data flow. 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 standard Java.

Our results therefore suggest that READABLE does improve program understanding. Hypothesis H1 was supported, and for practical purposes, hypothesis H4 (a control hypothesis) was rejected. Although hypotheses H2 and H3 were not supported, they did not generate countervailing evidence. Given that H1 (control-flow) is the most important program understanding construct, it can be argued that the experiment demonstrated that READABLE does facilitate program understanding. Furthermore, as the approach results in an executable application, documentation and development occur simultaneously. Thus, the approach does not interfere with the rapid development cycles preferred for Internet technologies. Maintenance programmers are assured that overall system documentation is available and is a current reflection of the actual web application.

## Conclusion

This research proposes an executable self-documenting approach for web applications called READABLE. In the approach, web components (e.g., HTML, Java Servlets) and their control-flows are represented as tuples in a relational database. A controller layer uses the tuples in the relational database to execute the web application. Maintainers can also query the relational database to understand the application.

An environment developed in Java was employed to create various web applications thereby demonstrating the feasibility of READABLE. An experiment was performed to determine if READABLE facilitated program understanding. The results demonstrate that READABLE is both useful and viable for creating self-documenting web applications.

READABLE has been designed as an approach for constructing new applications. Due to possibly high switching costs, it may be infeasible or impractical to convert existing web applications to READABLE. However, given the large demand for new web applications that will eventually have to be maintained, this limitation in READABLE'S applicability is not especially problematic. Devising methodologies for porting existing applications to READABLE is future research.

In addition, we are attempting to apply READABLE as a basic architecture to develop software repositories. READABLE contains a natural method for encapsulation, so it can be employed to develop variations of a class of components such as search algorithms, provided the architecture of that component class is similar. Components can then be automatically assembled to develop software applications such as search engines.

## *References*

Aoyama, M. “Web-Based Agile Software Development,” *IEEE Software* (15:6), 1998, pp. 56-65.

Atkins, D.L., Ball, T., Bruns, G., and Cox, K.C. "MAWL: A Domain-Specific Language for Form-Based Services,” *IEEE Transactions on Software Engineering* (25:3), 1999, pp. 334-346.

Baskerville, R.L., Levine, L., Pries-Heje, J., Ramesh, B. and Slaughter, S.A. “How Internet Software Companies Negotiate Quality,” *IEEE Computer* (34:5), 2001, pp. 51-57.

Berners-Lee, T., and Cailliau, R. “WorldWideWeb: Proposal for a HyperText Project,” 1990. **http://www.w3.org/Proposal.html.**

Boehm, B.W. “Software Engineering,” *IEEE Transactions on Computer* (25), 1976, pp. 1226-1241.

Brabrand, C., Moller, A., and Schwartzbach, M.I. “The <bigwig> Project,” *ACM Transactions on Internet Technology* (2:2), 2002, pp. 79-114.

Brooks, R. “Towards a Theory of the Comprehension of Computer Programs,” *International Journal of Man-Machine Studies* (39:2), 1983, pp. 237-267.

Cailliau, R., and Ashman, H. “Hypertext in the Web: A History,” *ACM Computing Surveys* (31:4es), 1999, pp. 1-6.

Cavaness, C. *Programming Jakarta Struts*, O'Reilly & Associates, Cambridge MA. 2002.

Chiang, R. H .L. “A Knowledge-Based System for Performing Reverse Engineering of Relational Databases,” *Decision Support Systems* (13:3-4), 1995, pp. 295-312.

Chua, C. E. H., Cao, L., Cousins, K., Mohan, K., Straub Jr., D. W., and Vaishnavi, V. “IS Bibliographic Repository (ISBIB): A Central Repository Of Research Information For The IS Community,” *Communications of the Association for Information Systems* (8:27), 2002a, pp. 392-412.

Chua, C. E. H., Storey, V. C., and Chiang, R. H .L. “A Software Engineering Environment for Search Engine Development,” *Proceedings of the 12th Workshop on Information Technology and Systems*, 2002b, pp. 205-210.

Cohen, J. *Statistical Power Analysis for the Behavioral Sciences*, Lawrence Erlbaum Associates, Hillsdale, NJ. 1988.

Cook, T. D., and Campbell, D. T. *Quasi-Experimentation: Design and Analysis Issues for Field Settings*, Houghton Mifflin Company, Boston, MA. 1979.

Douglis, F., Chapin, S. J., and Isaak, J. “Technical Activities Forum: Internet Research on Internet Time,” *IEEE Computer* (31:11), 1998, pp. 76-77.

Forward, A., and Lethbridge, T. C. “The Relevance of Software Documentation, Tools and Technologies: A Survey,” *Proceedings of the 2002 ACM symposium on Document engineering*, 2002, pp. 26-33.

Fraternali, P. “Tools and Approaches for Developing Data-Intensive Web Applications: A Survey,” *ACM Computing Surveys* (31:3), 1999, pp. 227-263.

Hair Jr., J. F., Anderson, R. E., Tatham, R. L., and Black, W. C. *Multivariate Data Analysis with Readings, Fifth edition*, Prentice-Hall, Upper Saddle River, NJ. 1998.

Johnson, S. C. “YACC: Yet Another Compiler Compiler,” In *UNIX Programmer's Manual*, 2, Holt, Rinehart, and Winston, New York, 1979, pp. 353-387.

Knuth, D. E. *Literate Programming*, Center for the Study of Language and Information, Stanford, CA. 1992.

Nørmark, K. “Requirements for an Elucidative Programming Environment,” *Proceedings of the 8th International Workshop on Program Comprehension*, 2000, pp. 119-128.

Parnas, D. L., Madey, J., and Iglewski, M. “Precise Documentation of Well-Structured Programs,” *IEEE Transactions on Software Engineering* (20:12), 1994, pp. 948-976.

Pennington, N. “Stimulus Structures and Mental Representation in Expert Comprehension of Computer Programs,” *Cognitive Psychology* (19), 1987, pp. 295-341.

Peters, D. K., and Parnas, D. L. “Using Test Oracles Generated from Program Documentation,” *IEEE Transactions on Software Engineering* (24:3), 1998, pp. 161-173.

Quatrani, T. *Visual Modeling with Rational Rose 2002 and UML*, Pearson Education, Reading, MA, 2002.

Rajlich, V., Doran, J., and Gudla, R. “Layered Explanations of Software: A Methodology for Program Comprehension,” *Proceedings of the Third Workshop on Program Comprehension*, 1994, pp. 46-52.

Robbins, J. E., and Redmiles, D. F. “Software Architecture Design from the Perspective of Human Cognitive Needs,” *Proceedings of the California Software Symposium*, 1996, pp. 16-27.

Soloway, E., and Ehrlich, K. “Empirical Studies of Programming Knowledge,” *ACM Transactions on Software Engineering* (10:5), 1984, pp. 595-609.

Storey, M.-A. D., Wong, K., and Müller, H. A. “How Do Program Understanding Tools Affect How Programmers Understand Programs?,” *Proceedings of the Fourth Working Conference on Reverse Engineering*, 1997, pp. 12-21.

Storey, V. C., Dey, D., Ullrich, H., and Sundaresan, S. “An Ontology-Based Expert System for Database Design,” *Data and Knowledge Engineering* (28:1), 1998, pp. 31-46.

Sun Microsystems “Javadoc Tool Home Page,” 2002. **http://java.sun.com/j2se/javadoc**

Ullrich, H., Purao, S., and Storey, V. C. "An Ontology for Classifying the Semantics of Relationships in Database Design," *Proceedings of the Conference on Natural Language in Database*, 2001, pp. 91-102.

van Deursen, A., and Kuipers, T. "Building Documentation Generators," *Proceedings of the International Conference on Software Maintenance*, 1999, pp. 40-49.

Visaggio, G. "Relationships Between Documentation and Maintenance Activities," In *5th International Workshop on Program Comprehension*, 1997, pp. 4-16.

Visconti, M., and Cook, C. "Software System Documentation Process Maturity Model," *Proceedings of the ACM Conference on Computer Science*, 1993, pp. 352-357.

von Mayrhauser, A. A., and Vans, A.M. "From Code Understanding Needs to Reverse Engineering Tool Capabilities," In *Procs. Sixth International Workshop on Computer-Aided Software Engineering*, 1993, pp. 230-239.