ICIC Express Letters
Volume **13**, Number **12**, December **2019**

ICIC International ©2019 ISSN 1881-803X
pp. **1157–1167**

SPREADSHEET AND COMMA-SEPARATED VALUES (CSV) BASED DOMAIN-SPECIFIC PROGRAMMING LANGUAGE FOR WEB APPLICATION DEVELOPMENT

Noprianto[1], Benfano Soewito[2], Ford Lumban Gaol[1]
and Bahtiar Saleh Abbas[1]

[1]Computer Science Department, BINUS Graduate Program – Doctor of Computer Science
[2]Computer Science Department, BINUS Graduate Program – Master of Computer Science
Bina Nusantara University
Jl. K. H. Syahdan No. 9, Kemanggisan, Palmerah, Jakarta 11480, Indonesia
nop@noprianto.com; { bsoewito; fgaol; bahtiars }@binus.edu

Received May 2019; accepted August 2019

Abstract. *From programming language point of view, when developing web applications, programmers may use domain-specific programming languages or general-purpose ones. Depending on the experiences and languages, developing a web application could be a simple or complex task. It is complex because the programmer should use more than one languages (HTML, CSS, Javascript, language at the server side, and others). To reduce the number of languages, programmers may use domain-specific languages. The more specific the language, the simpler the development could be. However, some researches show that language specificity itself could be another challenge. Therefore, to make web application development simpler and at the same time make the domain-specific language useful, more research is needed. We propose the usage of spreadsheet – software packages that have been around for about four decades. We emphasize simple modeling technique, combined with column/row based syntax, saved as comma-separated values (CSV) file format, which can be interpreted as a fully functional web application. Experiments prove this method is usable and results in simpler web application development with significantly less development time (43%) and lines of code (30%) compared with standard web technologies.*

Keywords: Domain-specific programming language, Spreadsheet, Comma-separated values, Web application, Blog

1. **Introduction.** One of the goals of the World Wide Web – introduced in the early 1990s – is to make it possible to access information from any source in a consistent and simple way [1]. Collections of documents might be easily accessed using standardized protocol and document format that can be implemented to work across computing platforms. The protocol is HTTP and the document format is HTML, that may contain links to another documents. HTTP protocol works using client/server model, where an HTTP client (or user agent, typically a web browser) requests some resources to an HTTP server. The server then gives responses if the requested resources are available.

In earlier implementation of the World Wide Web, the requested resource was static resource, i.e., existing file on a filesystem [1]. This eventually developed into dynamic resource when HTTP server softwares were capable of running some program and returned the output as HTTP response [1]. Because such program is not limited to return simple text information or opening existing files, more possibilities are open. For example, the program may open a connection to a database system, performs some database operation, and returns the result back to the HTTP server. All of these are transparent to users:

DOI: 10.24507/icicel.13.12.1157

they request some resources and the server returns some responses. This serves as the basics of web application as seen today.

While the basics sound simple, the mechanisms are not. How some programs are run by HTTP servers are not implemented in the same way. In earlier days, we typically worked with CGI (Common Gateway Interface). CGI was the first consistent server-independent mechanism [2]. As long as the program speaks HTTP protocol, it can be developed using any programming languages. While CGI is still being used, alternatives are available, such as template processing.

If we can freely choose the programming language, what makes the difference between one language and another? We believe that easy or hard to use is not the definitive answer because it probably depends on the programmer and the domain of the application. We suggest that the main difference is the expressiveness of the programming language, that affects the source code needed to develop a web application.

From the domain of the application, we can categorize programming languages into general-purpose and domain-specific. The former can be used to develop any – if not all – kind of applications. The latter – as the name suggests – can only be used to develop very specific or limited kind of applications. Domain-specific languages trade generality for expressiveness in a limited domain [3]. There are advantages and disadvantages of both language categories.

In no specific order, we may list the advantages of general-purpose languages as the following: libraries and tools available, application domain, and programmer's investment. General-purpose languages, particularly the popular ones, enjoy a lot of libraries and tools to ease the application development. This may significantly speed up the development [4]. In top programming languages, almost all of them are general-purpose [5]. However, when we talk about web application development, there are some challenges. If the programming platform does not come with mechanism to interface with HTTP servers, we may end up using CGI. Even when ignoring the computing resources needed, we typically need to code more to manually speak HTTP if we have to stick with CGI [2]. If the platform supports web application development, or some standards are supported, the implementation may require us to write HTTP-specific code – to some extent. After all, if a general-purpose programming language comes with very specific construct for web development at the core language level, it will eventually become domain-specific language.

On the other side, domain-specific languages offer solution directly targeted at the problem domain. They offer expressive power, focused on and usually restricted to, a particular problem domain [6]. For the web, a domain-specific language may contain everything needed to develop a fully functional web application in few lines of code, with less or no technical details on HTTP client and server, and everything between them. From security point of view, using a specific language may reduce the possibility of unsafe codes, as the underpinning operations are handled by language interpreter or compiler. Based on a recent research, some security-related problems might be caused by some programming styles [7].

Now we move to the disadvantages. A research already listed the disadvantages [8]: development cost, education cost, limited availability, difficulty of finding the proper scope, difficulty of balancing between domain-specificity and general-purpose construct, and possible loss of efficiency. Based on web application domain, we see other challenges: how to make the domain specific language easy to use, hide the technical HTTP details from the programmer, and – at the same time – provide simple modeling of the user interface/web page layout.

In this work, we propose new, spreadsheet-based domain-specific programming language to simplify the development of web applications. The proposal starts with related works, on the usage of spreadsheets as the modeling tool and programming language

semantics. In proposed method section, the details on how the language works are presented, including language syntax, semantics, and rules. For interested readers, we also provide the language interpreter along with discussion on its architecture. Experiments and their results are also presented, followed by conclusions.

This work contributes to web development research, in form of a domain-specific language, to simplify the development and free the programmers from lower-level details on HTTP, while producing some standard-compliant HTML codes.

2. **Related Works.** Based on our previous research on spreadsheet, we learned that spreadsheet-based modeling is helpful, mostly because of its table-based system [4]. Spreadsheet can also hide the programming complexities from the programmer, for example, eliminating the need to explicitly specify the data type. Its computational techniques match users' task and shield users from the low-level details of traditional programming [9]. And, for styling user interface, we can use the cell attributes [4].

In modern spreadsheet software, a single spreadsheet document may contain one or more worksheets. For some application, a single worksheet itself may contain up to million rows and thousand columns [10]. As a very big table, a worksheet offers more than enough cells to model a user interface or an application. The modeling is not limited to web application. We can also model the desktop GUI application or possibly another kind of user interfaces [4]. Even some GUI libraries come with similar concept with their grid-based layout manager [11, 12]. The key idea is to layout a user interface in a grid. Researches prove this method significantly reduces the time needed to manually code the user interface – for more than one platform in single code base [4].

Spreadsheet offers more than just a table-based interface. We can program without knowing too much about the detail. For example, we can put a number into a cell, without specifying the data type. We can put a string without the need to surround it with quotes. We can put values anywhere and reference them later. All of these can be combined with simple formula. Spreadsheet offers an environment suitable for many types of applications, from small scratch pad to complex ones [13].

A cell in a worksheet is more than just a value. After putting some text, we can style the cell. Styling may be as simple as applying text color or background color [4]. We may also format the text as bold, italics, and so on. If we combine the styling with user interface layout, we can end up with a colorful multi platform user interface, modeled by a single colorful worksheet [4].

The usage of spreadsheet to model a web application is not a new concept. A research from Benson et al. presented a system called Quilt that allows users to do rapid prototyping [14]. The design goals were to reuse existing HTML and spreadsheets as much as possible, minimizing the programming-like concepts. Another research showed how to create interactive web data applications with spreadsheet, presented a live programming environment that extends spreadsheet metaphor [15]. Based on our previous research on data entry domain-specific language, spreadsheet semantics can be beneficial, particularly for spreadsheet users [16].

Compared with previous studies, this work presents the programming language and the modeling tool, using only a single spreadsheet document to develop a fully functional database-backed web application.

3. **Proposed Method.** Every modern spreadsheet software may suggest a default file format. That way, a spreadsheet software can put extra features, probably specific to them. Today, it is common that one spreadsheet software can open from or save to another file format used by another spreadsheet software, with some level of compatibility [17]. To make this situation better, some file formats are standardized and their specifications are open.

While it seems that standard-based file formats are good for spreadsheet usage, they are probably too complex for simple programming. Our main consideration is the binary format that requires us to use the spreadsheet software to edit the file. We like the spreadsheet for modeling (mainly for user interface) and coding, but for simple programming or revision, we prefer textual format. In some environments, for example at text-based server where GUI is not available, running graphical spreadsheet software is not possible. If only text-based text editors are available, textual format is the natural choice.

In addition to their native file format, many modern spreadsheet softwares support exporting to text-based representation of the tabular data. Usually, the textual format would be CSV file format, that defined in RFC 4180. It is text-based [18], can be edited by any text editor, and sufficient for our domain-specific language.

Our proposed language targets the development of a web application. To make it easier to measure, we focus on web log (blog) application. In a blog, there is a user interface, where a series of post [1] and some static contents are displayed (a sample is illustrated in Figure 1). Technically, these contents are saved in a database management system or some files in a filesystem. Therefore, other than modeling the user interface, we must deal with reading and writing to some external resources. Using a single CSV file, edited in spreadsheet software, our proposed language provides everything from blog layout design to simple programming construct.

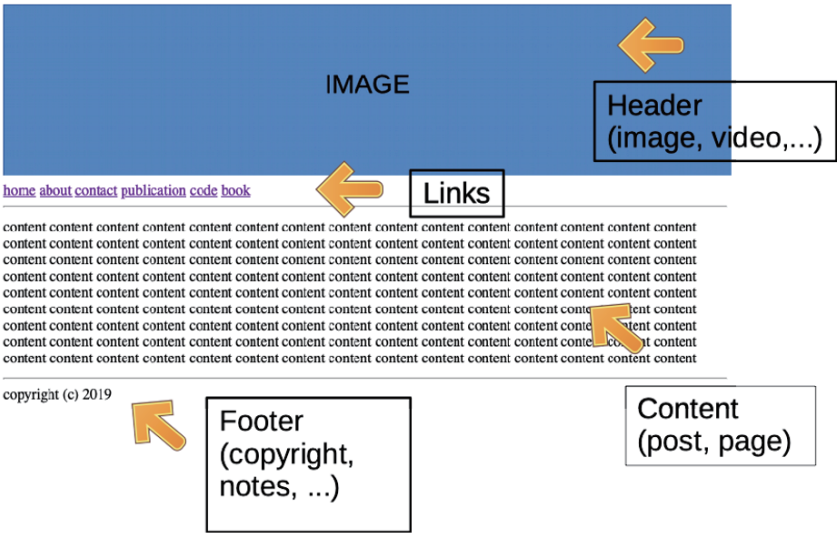


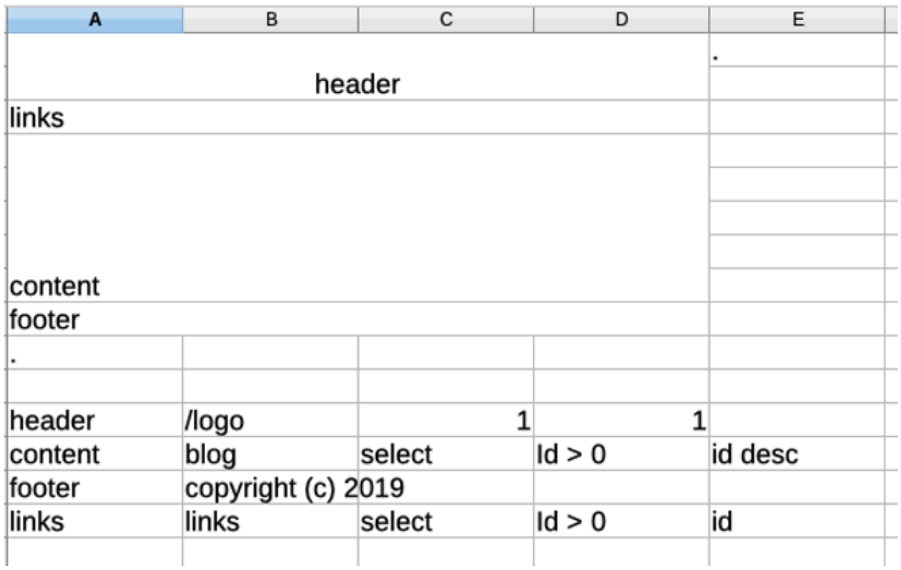
FIGURE 1. A sample of web log (blog) user interface

For the user interface part, we propose that the layout is mapped to worksheet column and row as our previous research [4]. However, we advance to allow cell merging because this feature is common and available in many spreadsheet softwares. More importantly, cell merging allows the programmer to layout closely to the final result.

Based on our internal discussions, it is easier to rule where the layout should be started and positioned. In this case, we propose that the layout must be started at cell A1. Sample of the layout (with some additional codes) is illustrated in Figure 2.

It is worth noting that we do not use reserved words in our language. The layout may consist of header, content, links, and footer, but these texts are not reserved words. Page header can be written as header, X, Y, or other literals. Our interpreter will simply treat them as text literal if the definition for that symbol is not found.

Cell merging is actually challenging when it comes to CSV format. In Figure 2, there is a layout where a header is spanned to 4 columns and 2 rows. In spreadsheet software, it is clearly seen and understood. However, without additional content below the table, for example, the saved CSV only has one row and one column. This is understandable



A	B	C	D	E
				.
header				
links				
content				
footer				
.				
header	/logo	1	1	
content	blog	select	Id > 0	id desc
footer	copyright (c) 2019			
links	links	select	Id > 0	id

FIGURE 2. Simple page layout with cell merging

because cell merging information is not saved into the CSV format. After all, all we have is a single cell. That is why we have to use some auxiliary symbol to inform the interpreter that there is a cell merging near specific cells. We tested this idea to our tester, and we finally came up with a dot (.). In our proposal, a dot may semantically represent a stop or an end. Below the merged cells, we put a dot to represent that we end that definition. The resulted CSV honors that and put number of rows as spanned. The dot is only needed when there is no additional contents below or after the merged cells. If we put content below the merged header, for example, content, the dot is not needed. All we need is an extra character. Figure 2 illustrates the supported layout definition with some merged cells.

Let us move to the almost none reserved words rule. The rule is simple: if a text (or a symbol) is not defined later, treat it as a text literal, not a variable or a command. Definition means the text is put somewhere else, with some parameters given. For that purpose, one can use spreadsheet formula = [CELL]. For example, if A1 cell contains header text, and that symbol is intended to be a variable or a command, the definition may be put in another cell using = A1 formula. That way, a symbol is not limited to a single word or a single character: they could be a very long text (with whitespaces), as allowed by spreadsheet software.

We also propose that our language treats all symbols as string, more or less found in Tcl programming language [19], although our proposal is certainly much simpler and only scoped to web application domain. One may put some values such as 1, 2, x, y, z, and we will treat them all as strings. Our intention is to make the language simpler and to align with simple programming model offered by spreadsheet [9].

Treating all symbols as string in our proposal means that we have no predefined or strict data type. If a text looks like a number and the context allows a number, we will try to convert the text into its numerical value and act appropriately when the conversion fails. A context is a combination of symbol definition and where the symbol applied. One of the best examples would be in looping. One can put text 5 in a cell for looping. Because the scope is looping and the language supports only ranged loop, the interpreter will automatically convert the value into numeric.

As we avoid data type and reserved words (other than the dot), it was relatively tricky to support looping or conditional statements. There is no 'if' syntax because 'if' is simply a text literal. Or, it could be a variable or a command if it defined so. Should we use

if function (= if) which commonly supported by modern spreadsheet software? Unfortunately, in this case, we decided not to use spreadsheet functions. The main reason is functions availability that could be different between a spreadsheet software and another. Another reason is to avoid syntax or logic error in the usage of spreadsheet function (and make the language prone to error and harder to use). We learn from some researches that point to errors in spreadsheets. Originally, spreadsheets were meant for domestic usage. However, time has proved that spreadsheets are widely used than was anticipated [20]. Errors are common, with rate from one percent or more of all formula cells [13].

That decision led us to use another construct: put a symbol in a cell and put the parameters next to the symbol (at the right side). These parameters – along with the context – will decide what action should be taken. Details of supported actions are listed in Table 1. Figure 2 illustrates the usage of some actions.

Table 1. Supported actions

Context	Action	Param 1	Param 2	Param 3	Param 4
any	looping	start	stop	step	cmd
any	conditional	cmd (test)	cmd (if test is true)		
any	variable definition	value			
any	command definition	param	param	param	
any	database operation	table	operation	filter	order

From Table 1, we can see that for available actions, context is not used in current design and probably be used in future developments. Here are details for each action.

- Looping: each of start, stop, or step should be convertible to numeric. Command should be defined. If conversion fails or command is not defined, respective action will not be performed.
- Conditional: will be interpreted as simple if test. If a command returns something evaluates to true (parameter 1), a command will be run (parameter 2). Both commands must be defined, otherwise the respective action will not be performed. There is no if-then-else construct.
- Variable definition is simple and takes only one parameter: the value. If the value is not specified, an empty value will be assigned to the variable.
- Command definition needs more explanation because it is the most complex one. Firstly, all of these three parameters are not strictly defined. Secondly, some parameters may return something or nothing. And lastly, any parameter may be unused, but a command must be defined with three parameters. To make it useful, some parameter may be an operator (>, >=, <, <=, =, <>). Because any of its parameters may be unused or undefined, a command may not be run and left as is, without any syntax errors.
- Database operation takes four parameters. First parameter is the database table. Second parameter is one of: select, insert, update, or delete. Third parameter is used for filtering (where clause). Last argument is used for ordering, if applicable. Interpreter should ignore an operation that is nonsensical.

Some commands also have predefined behaviors. For example, in header defined in Figure 2, values in parameters (such as /logo, 1, and 1) are detected whether they matched some HTML semantics. For example, if it looks like /logo is an image declaration (combined with another parameters), it will be interpreted as `img` HTML tag. That way, this language may add additional features without breaking existing codes.

Because our language is simple and very specific to web application development, we suggest that the following additional rules may make the language easier to use:

- Case sensitivity: case-insensitive. That way, programmers do not need to exactly remember the case of a symbol. For example, header and HEADER are both refer to the same symbol.
- Variable name: must not be a number and whitespace-only. Length limit is not specified.
- Action definition must be outside the page layout table.
- Database name and connection properties are not defined and are implementation-specific.
- For each row in user interface table, only one component (single cell or merged cells) is supported. If more than one components are defined, it is implementation-specific and might be ignored.

4. **Interpreter Implementation.** We provide an implementation as an interpreter, integrated into SQLiteBoy (http://sqliteboy.com, https://github.com/nopri/sqliteboy). While SQLiteBoy was originally a web-based management tool for SQLite database, it also comes with form, report, and website features. The interpreter is integrated as a part of SQLiteBoy, starting from version 1.74. SQLiteBoy is a free/open source software, written in Python, in development since 2012, and consists of only a single source file (13,000+ lines of code, version 1.74).

Because underpinning operations are handled by SQLiteBoy, our main work is mainly divided into: user interface table recognition, symbol definition handling, and web page generation. Scanning and parsing steps are done using CSV file parsing algorithm and they are well served by Python CSV module.

For user interface table recognition, we can always be sure that the table is started in first column and first row. What we care is the end of the table and how to deal with cells. This is the algorithm used in recognizing the table and interpreting cells and merged cells in each row.

- We iterate non-empty cells both in columns and rows to look for the dot characters. For columns, first dot character denotes the width of the table. For rows, if the dot character is found, row number represents the height of the table. If no dot characters are found, recognition is done based on last non-empty cells in all columns and rows. This might take time and this action is implementation-specific. Because this behavior is not defined in the specification, it might be changed in future versions.
- After we get the boundaries, we convert the table into list-of-list data structure representation. Some cleanup actions are performed, such as ignoring columns in a row if the column number is more than one (only first column in a row is read) and padding the list if number of columns/rows is not equal (maximum values are used).
- We iterate the list to find merge cells and build another list structure. For each symbol found, we iterate its next rows until we find a row containing another symbol. Number of empty rows in-between is the extra number of rows occupied in the user interface grid. At the end of this step, we have a list (named user_interface) consisting of [<symbol>, <extra rows>] list items.

For symbol definition handling, we read the CSV file outside the range of user interface table. For each symbol found in user_interface list, we need to perform clean up action by ignoring invalid action (based on number of parameters). This step results in a hash table/dictionary (named symbols).

For web page generation, we use the information from user_interface list and symbols dictionary. Main HTML output is performed in an iteration. For each symbol in the user_interface list, if such symbol is defined in the dictionary, interpretation is done based on action category. Otherwise, treat the symbol as a text literal. Interpretation might involve another table lookup in the symbols. Nonsensical operations are also ignored.

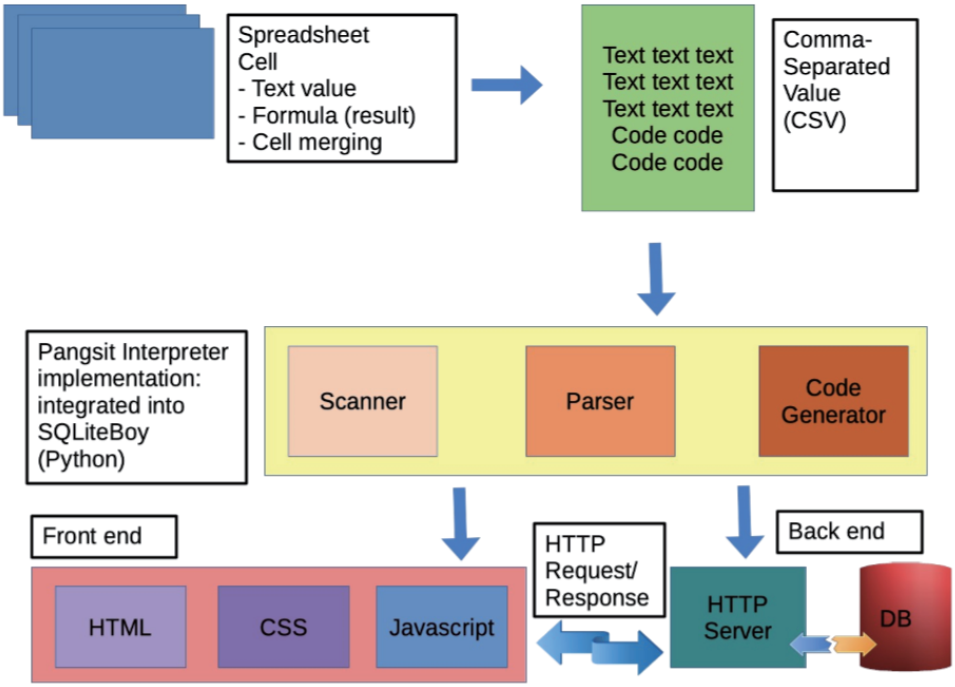


FIGURE 3. Language interpreter architecture

Additionally, as noted, some commands have predefined behaviors and they will determine the output.

How the language interpreter works is illustrated in Figure 3. From a spreadsheet document, the tabular data are saved into a CSV file. Through an interpreter integrated into SQLiteBoy, front-end and back-end of a web application are generated and/or processed. The front-end consists of HTML, CSS, and Javascript, while the back-end consists of server side codes, backed by a database system.

5. **Experiment and Result.** As a specific language that is based on spreadsheet, firstly we would like to measure the language itself for experienced spreadsheet users. By experienced, such users must use spreadsheet on a daily basis and have been using spreadsheet software for at least 3 years. However, no programming experience is assumed. We invited 10 respondents and tasked them to define a table (with cell merging) to model a web page. After that, we also tasked the user to define some simple actions. We measure the time needed and number of errors. Table 2 lists the results (first round).

TABLE 2. Time needed and number of errors using spreadsheet to develop a web application (first round)

Respondent	Design time (minute)	Number of errors
1	10	3
2	8	2
3	15	4
4	8	1
5	8	2
6	12	0
7	15	0
8	7	1
9	7	2
10	13	2

Table 3. Error type, fix, and language change (based on previous experiment)

Error	Fix	Language change
Number as variable name	None, user must change the variable name	None
Use of reserved word for web page section (typo, cannot remember the word)	Language change	No reserved words, except for dot symbol
Use of spreadsheet functions (syntax error, unknown function)	Language change	Use parameters
Actions defined in layout table	None, user must do that outside the table	None
Use of operator for text value	Language change	Allow the operator to be applied to text value by using its length property
Errors related to database operations	Language change	Ignoring nonsensical operations

Table 4. Comparison with other programming languages

Respondent	Programming language	Development time (minute)	Lines of code
1	PHP	30	44
2	PHP	40	39
3	Java (Servlet)	80	180
4	Java	70	120
5	Python (Standard libraries)	40	50
6	Python (Framework)	35	40
7	Python (Micro framework)	30	42
8	Perl (CGI)	40	46

We set the time needed target as 30 minutes. Because we did not reveal this information and all of the respondents finished the task quicker, we conclude that their elapsed time was acceptable. We were more interested with the number of errors. The errors were reported by the simple validator and mostly related with programming activities. Table 3 lists the errors.

In order to make the language suitable and usable by regular spreadsheet user, we changed the language appropriately to what become our proposal. We then repeat the experiments until we finally achieved maximum time needed of 13 minutes and maximum number of errors of 1. All done in 12 to 15 lines of code (number of rows). All of these results were very acceptable to us, to allow us to continue with another experiment.

Based on the result, we would like to measure how the proposed language is compared to other programming languages. We concentrated only about the basic web page generation and simple user interface, and did not measure the codes needed for underpinning operations such as database connection or authentication. For that purpose, we invited 8 computer programmers who are familiar with web application development. They chose their favorite language/framework and we measured development time (in minute) and lines of code, as listed in Table 4.

We measure lines of code because based on some research, lines of code contributed to number of bugs and software maintainability. Number of bugs vary, for example 6-16 bugs per 1000 lines of code [21]. Also, we would like to measure whether our language offers significant less effort. In this case, compared with shortest development time (30

minutes), our language offers significantly less time (13 minutes) with 43% ratio. Similar comparison result was found in lines of code: 12 compared with 39 lines of code (30% ratio).

Our latest comparison was to measure how the generated web page complies with HTML standards. Several reasons are listed [22]. One of them could be summarized as the more the generated web page comply with the standards, the better it may render on more web browsers. Using the results from previous respondents, we used an external validator service to validate whether generated web pages were standard-complaint. The result was only 1/4 (25%) that complies with HTML standards. Our language offers near 100% HTML standard compliant because we strongly endorse this.

6. **Conclusions.** Based on the experiments and results, our proposed method is usable and useful in web application development (for blog application). Our spreadsheet-modeled proposal offers significantly less time and lines of code, while at the same time, resulted in HTML standard compliant codes. The usage of CSV format also makes it easier to edit using only a text editor. For seasoned developers, it is also possible to fully develop using only a text editor.

However, we do aware of its limitations. Compared with our research on multi platform user interface modeling, we lose the usage of cell attributes to style the web page. We addressed the cell merging, but lost the cell attributes and multi worksheets when we switched to CSV file format. Another method for web page styling is certainly needed. Additionally, while cell merging is supported, each row in user interface table can only contain one component.

While we provide simple command, looping, and conditional statement, they are probably quite basic. For example, only one looping construct is provided. There is also no if-then-else construct. Commands are also limited. At current design and implementation, a simple web application could be developed. However, further development may require more advanced features.

REFERENCES

[1] M. Jazayeri, Some trends in web application development, *Proc. of Future of Software Engineering (FOSE'07)*, pp.199-213, 2007.
[2] L. S. R. Rosen and L. Shklar, *Web Application Architecture: Principles, Protocol and Practices*, John Wiley, Hoboken, NJ, 2009.
[3] M. Mernik, J. Heering and A. M. Sloane, When and how to develop domain-specific languages, *ACM Computing Surveys (CSUR)*, vol.37, no.4, pp.316-344, 2005.
[4] Noprianto, B. Soewito, F. L. Gaol and B. S. Abbas, Multiplatform application user interface design based on spreadsheet, *AdMITS*, 2017.
[5] *Tiobe Index*, https://www.tiobe.com/tiobe-index/, Accessed on 09-Mar-2019.
[6] A. Van Deursen and P. Klint, Domain-specific language design requires feature descriptions, *Journal of Computing and Information Technology*, vol.10, no.1, pp.1-17, 2002.
[7] A. Kurniawan, B. S. Abbas, A. Trisetyarso and S. M. Isa, Classification of web backdoor malware based on function call execution of static analysis, *ICIC Express Letters*, vol.13, no.6, pp.445-452, 2019.
[8] A. Van Deursen, P. Klint and J. Visser, Domain-specific languages: An annotated bibliography, *ACM Sigplan Notices*, vol.35, no.6, pp.26-36, 2000.
[9] B. A. Nardi and J. R. Miller, The spreadsheet interface: A basis for end user programming, *Hewlett-Packard Laboratories*, 1990.
[10] *Frequently Asked Questions – Calc*, https://wiki.documentfoundation.org/faq/calc/022, Accessed on 09-Mar-2019.
[11] *Gtkgrid: Gtk+ 3 Reference Manual*, https://developer.gnome.org/gtk3/stable/gtkgrid.html, Accessed on 09-Mar-2019.
[12] *Qgridlayout Class*, https://doc.qt.io/archives/qt-4.8/qgridlayout.html, Accessed on 09-Mar-2019.
[13] R. R. Panko, What we know about spreadsheet errors, *Journal of Organizational and End User Computing (JOEUC)*, vol.10, no.2, pp.15-21, 1998.

[14] E. Benson, A. X. Zhang and D. R. Karger, Spreadsheet driven web applications, *Proc. of the 27th Annual ACM Symposium on User Interface Software and Technology*, pp.97-106, 2014.
[15] K. S. P. Chang and B. A. Myers, Creating interactive web data applications with spreadsheets, *Proc. of the 27th Annual ACM Symposium on User Interface Software and Technology*, pp.87-96, 2014.
[16] Noprianto, B. Soewito, F. L. Gaol, B. S. Abbas, S. W. H. L. Hendric and A. Trisetyarso, Perkedel: Spreadsheet-inspired domain-specific programming language for data entry, *IEEE CyberneticsCom*, 2017.
[17] *File Formats – Apache Openoffice Wiki*, https://wiki.openoffice.org, Accessed on 09-Mar-2019.
[18] *RFC 4180*, https://www.ietf.org/rfc/rfc4180.txt, Accessed on 09-Mar-2019.
[19] *Everything is a String*, https://wiki.tcl-lang.org/page/everything+is+a+string, Accessed on 09-Mar-2019.
[20] M. Csernoch and P. Biró, Spreadsheet misconceptions, spreadsheet errors, in *Belvedere Meridionale*, Szeged, 2014.
[21] A. S. Tanenbaum, J. N. Herder and H. Bos, Can we make operating systems reliable and secure?, *Computer*, vol.39, no.5, pp.44-51, 2006.
[22] *Why Validate?*, https://validator.w3.org/docs/why.html, Accessed on 09-Mar-2019.