Journal of Visual Languages and Computing (2001) **12,** 435–472
doi:10.1006/jvlc.2000.0198, available online at http://www.idealibrary.com on IDEAL®

Visual Programming in the Wild: A Survey of LabVIEW Programmers

K. N. WHITLEY* AND ALAN F. BLACKWELL†‡

**Department of Computer Science, Vanderbilt University, Box 1679, Station B, Nashville, TN 37235, U.S.A., e-mail: whitley@computer.org and †Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, U.K., e-mail: alan.blackwell@cl.cam.ac.uk*

Received 30 April 1998; revised 30 September 2000; accepted 22 November 2000

As part of research into the cognitive effects of visual programming representations, a worldwide survey of LabVIEW programmers was conducted via the World Wide Web and resulted in 227 survey responses. The survey questionnaire was designed to allow some distinction between opinions about the visual aspects of LabVIEW and opinions about LabVIEW's many other features; this paper presents an analysis of this collected data, including both the opinions of the visual and non-visual aspects. Throughout the analysis, comparisons are made from the survey data both to Green's cognitive dimensions and to Green and Petre's own assessment of LabVIEW [1, 2]. Overall, the respondents were positive in their assessment of LabVIEW. Moreover, of interest to the visual programming community, respondents rated the value of LabVIEW's visual language significantly higher than the value of all other LabVIEW features rated in this survey (including LabVIEW's extensive libraries of reusable code). At the same time, respondents' comments suggest many other possible sources of LabVIEW's success such as LabVIEW's libraries of reusable code, LabVIEW's support for building GUIs, LabVIEW's use of the dataflow paradigm and LabVIEW's automatic memory management. By identifying these non-visual aspects of a VPL, this research hopes to contribute to future, rigorous investigation of the impact of visual programming representations.

Keywords: empirical evaluation, cognitive dimensions, LabVIEW.

1. Introduction

HOW CAN WE FIND good directions for visual programming language (VPL) research? One perspective is that transforming graphics into computational structures is interesting as pure theory. There is an alternative perspective–that VPLs are significant because they promise improvements in the usabililty and accessibility of programming tools. If the second factor is relevant, research must be directed in part by an understanding of VPL users: what do they need in a VPL and how would they use it? VPL researchers

‡Alan Blackwell's research was funded by a collaborative studentship from the Medical Research Council and Hitachi Europe Ltd. He is grateful to the Advanced Software Centre of Hitachi Europe for their support.

1045-926X/01/040435+38 $35.00/0

have often neglected this issue. They make assumptions about user requirements based on preconceptions and folk wisdom, rather than scientific observation [3]. We propose that VPL research be partly directed by methodical study of skilled users who are experts in the use of existing VPLs. In the research reported in this paper, we demonstrate this approach by example.

This paper reports the results of an exploratory survey of experienced LabVIEW users, a main goal being to learn how LabVIEW programmers think that the visual representation of LabVIEW affects their programming. This research is a step towards understanding how visual representation impacts the cognitive processes of programmers. Pursuit of this goal requires careful attention to definitional issues; in order to uncover effects accruing from a visual representation, questions arise about what constitutes the visualness of the representation. Particularly when studying visual representation in the context of a fully implemented VPL, a researcher must distinguish between the visualness of the VPL and its potentially confounding, non-visual features.

As an exploratory survey, the results presented herein provide grist for subsequent research. In general, the results are limited in that people's perceptions of their cognitive processes are not necessarily accurate; thus, a person's claims as to how and why a VPL impacts the programming process cannot be accepted as fact without further empirical study. Moreover, the LabVIEW survey respondents were self-selected and, so, cannot be assumed to be representative of all LabVIEW programmers. However, their opinions are based on real experience using LabVIEW and, as a group, can provide a more complete understanding, especially if balanced with controlled studies. The value of the survey stems from both its design and details of the collected opinions. The care taken in its design to distinguish between the visual and non-visual features of LabVIEW illuminates the issues that can potentially confound studies of VPLs. The details of the respondents' opinions provide specific directions for further research. For example, in considering the collected comments about LabVIEW's use of the dataflow paradigm, the first author has subsequently completed a controlled experiment investigating the effects of representation independent of paradigm [4]. In these ways, this survey promotes the methodical study of the effects of VPLs in order to replace the VPL community's prior assumptions about how VPLs affect programmers.

The questionnaire designed for the LabVIEW survey asked for opinions about a wide range of LabVIEW features and allowed some discrimination between LabVIEW's visual and non-visual aspects. The questionnaire included open-format questions about LabVIEW as a whole and one question that specifically asked LabVIEW programmers about how visual programming affects their thought processes. The responses collected from these open-format questions have been grouped into categories, with care given to noting when the respondents referred to visual versus non-visual aspects of LabVIEW. Additionally, the questionnaire included ratings tables; one table requested respondents to rate the advantages of various LabVIEW features, while another asked respondents to compare the advantages of textual and visual programming along several dimensions. The data collected from these ratings tables have been statistically analyzed.

Overall, the respondents were positive in their assessment of LabVIEW. Of interest to the visual programming community, respondents rated the value of LabVIEW's visual language significantly higher than the value of all other LabVIEW features rated in this survey (including LabVIEW's extensive libraries of reusable code). At the same time, respondents' comments suggest many other possible sources of LabVIEW's

success. Namely, respondents commented on a wide array of LabVIEW features including LabVIEW's libraries of reusable code, support for building GUIs, use of the dataflow paradigm and automatic memory management.

A preliminary report of the LabVIEW survey results appears in a paper comparing the results of three opinion surveys [5]. All three surveys collected beliefs about the cognitive effects of visual programming. The first examined the visual programming literature for the opinions of academic researchers. The second gathered the opinions of professional programmers attending a trade show. Rounding out the trio, the LabVIEW survey targeted experienced users of LabVIEW, one of the more successful commercially available VPLs. This paper presents the findings of the LabVIEW study. Due to space considerations, this paper presents an abridged version of the findings, especially in the reporting of the open-format data; interested readers can also consult the comprehensive report [6].

The remainder of this paper is organized as follows. Section 2 summarizes two empirical studies in the visual programming literature that are directly relevant to the LabVIEW survey. Section 3 focuses on the methods used in this survey by describing the questionnaire's design, survey administration, expected results and the rules used to analyze the open-format responses. Section 4 presents quantitative analysis, mainly of the data from the ratings tables. Sections 5 and 6 present the qualitative analysis of the open-format questions; these detailed sections are appropriate for language designers or readers designing experiments of visual programming representations. The visual aspects are summarized in Section 5, followed by the non-visual aspects in Section 6; throughout both sections, the survey results are linked to Green's cognitive dimensions [1]. Finally, Section 7 summarizes the overall findings.

2. Background

Laboratory Virtual Instrument Engineering Workbench (LabVIEW) is a programming environment that features a dataflow-based VPL (called G) designed to facilitate development of data acquisition, analysis, display and control applications. One of LabVIEW's marketing claims is that LabVIEW is so usable that it is an effective tool not only for trained programmers, but also for certain types of end users. In particular, LabVIEW is described as usable by scientists and engineers who possess limited programming experience, yet who need software to interact with laboratory equipment. One of the few commercially available VPLs, LabVIEW has enjoyed relatively wide success.

LabVIEW is an attractive choice for an empirical study of visual programming for several reasons. The fact that LabVIEW has been commercially available for over 10 years means that one can find a sizable population of people who have used it. Thus, given the desire to study an expert population, LabVIEW becomes a natural candidate. Moreover, within the fairly small number of empirical studies that have investigated visual representations in programming, two prior studies of LabVIEW pose some interesting questions (to place these two within the context of other empirical studies of visual programming, see [7]).

One of these studies is an industry-based, observational study reported by Baroth and Hartsough which describes their organization's experience using two VPLs (LabVIEW and VEE) to build test and measurement systems [8]. Of particular interest is their case

study, which compared the progress of two teams (one using LabVIEW and the other using the textual language C) that were, in parallel, each tasked with developing the same system. The goal was to discern how LabVIEW would compare to conventional, text-based programming. Both teams received the same requirements, funding and amount of time (3 months) to complete the project. At the end of the allotted time period, the C team had not achieved the original project requirements, yet the LabVIEW team had gone beyond the requirements.

From this case study and from experience using LabVIEW and VEE on over 40 other projects over the course of 3 years, Baroth and Hartsough enthusiastically report performance benefits for both VPLs. They report that projects using LabVIEW and VEE require 4–10 times less time to complete than if attempted using a textual programming language. Moreover, Baroth and Hartsough attribute the benefits of the two VPLs, in large part, to the visual representations of the VPLs. They claim that LabVIEW's visual representation is more readable for certain classes of end users than traditional textual programming languages. Because LabVIEW's visual syntax produces programs resembling wiring diagrams, Baroth and Hartsough claim that LabVIEW is relatively easily learned by scientists and engineers who are familiar with wiring diagram notations.

In contrast, a controlled study by Green *et al.* revealed no benefits resulting from LabVIEW's visual representations for conditional logic (LabVIEW provides two ways to represent conditional logic) [9, 10]. Their experiment compared the comprehensibility of LabVIEW's conditional representations to two textual representations by measuring response time required to answer questions about code segments. Their subjects were 11 experienced programmers. Five had used LabVIEW in their work for at least 6 months, while the other six were advanced digital electronics designers and, thus, experienced in using electronics schematics. LabVIEW elicited worse performance (i.e. longer response times) on all of the comprehension questions. In fact, the textual representations outperformed LabVIEW for each and every subject.

The 'contradictory' results obtained in the Baroth and Hartsough study and the Green *et al.* work motivated the LabVIEW survey reported here. The studies cannot be truly compared, due to the sheer difference in the scales of the studies. Yet, they do present the obvious question: what accounts for the difference between the positive and negative reports about LabVIEW's visual representation for the same class of end users? Several explanations could account for this gulf. For example, LabVIEW's visual representation for constructs other than conditional statements may account for the advantage reported by Baroth and Hartsough. Another possibility, however, is that Baroth and Hartsough were incorrect in attributing performance benefits to LabVIEW's visual representation. It may be that LabVIEW does improve programming but that the cause of the improvement stems from language features other than the visual syntax.

3. Methods

3.1. Materials

The LabVIEW survey was administered using the questionnaire shown in Appendix A. The first part of the survey collected information about a respondent's programming

background. Question 1 allowed respondents to give contact information, which was used to award incentive prizes (books, t-shirts and mouse pads supplied by National Instruments, the maker of LabVIEW) to three randomly selected respondents. Question 2 inquired about the nature of a respondent's programming activity. This question allowed differentiation between three groups: professional programmers; people who are not hired as programmers, *per se*, yet whose job involves some programming (i.e. end users); and people who teach computer science/programming. Question 3 sought a qualitative measure of respondents' programming experience, in terms of the size and number of projects completed both in LabVIEW and other languages. Question 4 asked respondents to name the programming language that they were most experienced in using.

The second part of the survey sought to elicit a respondent's opinions about LabVIEW as a whole, without emphasizing the visual aspects of LabVIEW. Questions 5, 7 and 8 were open-format questions that asked respondents for their overall opinion of LabVIEW, examples of how LabVIEW makes programming easier, and examples of how LabVIEW makes programming difficult. Question 6 specifically asked respondents to assess the value of six LabVIEW features using a six-point rating scale. The listed features of LabVIEW were as follows. The entry 'LabVIEW Toolkits/Reusable VIs' refers to the fact that National Instruments supplies LabVIEW buyers with reusable libraries of LabVIEW code; moreover, LabVIEW programmers can also acquire more VIs (this stands for 'virtual instruments', which is LabVIEW's term for software components) from additional sources. The entry 'National Instruments Hardware' refers to the range of data acquisition and control hardware available from National Instruments. The entry 'LabVIEW's Graphical Language (G)' refers to the visual representation offered by LabVIEW. The entry 'G's Use of Dataflow Programming' refers to the fact that, in addition to using visual representation, G is based on the dataflow paradigm. The entry 'LabVIEW's Support for 'Front Panel' Interfaces' refers to the fact that LabVIEW programmers can often easily create (without writing any code) the graphical user interface to their systems. Finally, the entry 'Customer Service for LabVIEW' refers to National Instruments' customer support facilities.

The third section of the survey asked explicit questions about the visual aspects of LabVIEW's VPL G. In Question 9, respondents were asked to compare G to a textual programming language of their choice along seven dimensions: power, writeability, readability, modifiability, enjoyability, whether the language adequately supports repetitive logic (i.e. the writing of loops) and whether the language adequately supports conditional logic (i.e. the writing of if–then–else statements). The power dimension was intended to elicit popular opinions of the kind often expressed by programmers, despite the fact that power in a programming language is not clearly defined. One can hypothesize that power is an assessment of effectiveness that includes issues of conciseness, flexibility and the programmer's access to low-level facilities of the hardware and operating system. Question 10 invited respondents to comment on the cognitive effects of G; this was achieved by asking respondents to explain how and why the graphical nature of G affects the 'brain-work' required in programming.

3.2. Procedure

The questionnaire was administered electronically using two versions, a WWW version and an e-mail version. Data collection for this survey focused on two main sources of

LabVIEW users. The larger source was info-labview, a mailing list for LabVIEW programmers. As of January 1997, info-labview had approximately 2300 subscribers. The second source was a list of 104 e-mail addresses compiled by National Instruments of academics who use LabVIEW either in conducting their research or in teaching their classes. Data collection took place in a 3-week period in March, 1997, during which time 227 complete responses were gathered.

3.3. Hypotheses

While the survey was primarily exploratory, the prior studies of LabVIEW motivated two specific questions about whether the visual representation used in G actually affords any benefits to its programmers. First, the degree to which LabVIEW fosters reuse by providing LabVIEW customers with repositories of pre-fabricated software components offers a compelling explanation that could account for LabVIEW's popularity and, thus, Baroth and Hartsough's results. Given the extent of available LabVIEW code libraries, programming in LabVIEW may largely consist of a search phase (in which the programmer looks for a software component that closely matches his or her needs), followed by a customization phase. This situation could account for LabVIEW's success, both with trained programmers and end users. An end user might be empowered by LabVIEW programming precisely because LabVIEW obviates the need for much programming.

Hypothesis 1. Even if the LV programmers rate the visual aspects of G as advantageous, they will rate the value of the reusability afforded by LabVIEW more highly than they will rate the visual aspects of G (in Question 6 of the survey).

Second, Green *et al.*'s findings suggest that LabVIEW's programmers may perceive some problems with LabVIEW's provisions for conditional logic.

Hypothesis 2. In Question 9, respondents will rate G more poorly than textual programming for the dimension assessing how a language provides for conditional logic.

3.4. Open-Format Responses: Coding Method

The qualitative analysis of responses to the open-format questions required breaking response text down into constituent opinions and, then, grouping the opinions into one of a list of semantic categories, or *themes*. To establish the themes, an initial reading of all of the text showed the variety of opinions present in the data; in fact, since the LabVIEW survey was analyzed in conjunction with the aforementioned surveys of the visual programming literature and of professional programmers, this step included an examination of all three data sets. The result of this process was a set of 16 *visual* themes, which apply to all three surveys, and a set of 13 *non-visual* themes, which are used solely in the LabVIEW survey. These are described in Sections 5 and 6, respectively.

After the themes were established, the second step was to assign each distinct opinion to one of the theme codes. Each LabVIEW survey submission was analyzed as a whole, meaning that all five open-format questions were coded using all of the themes. To

distinguish whether an opinion referred to the visual representation of LabVIEW, a conservative decision rule was enforced. Only those statements in which it was very clear that the respondent was referring to a visual aspect of LabVIEW received one of the visual theme codes; unattributed effects were coded with a non-visual theme.

Each opinion was also assigned a second code, marking it as *positive* or *negative*. For example, if a respondent made an assertion about the productivity impact of LabVIEW, the second code would reflect whether the respondent was claiming a productivity increase (a positive statement) or decrease (a negative statement). This coding dimension also included two other codes that were rarely used. A *conditional* code applied to undetailed opinions expressing a conditional outcome, for example 'It depends on the user's intended purpose'. An *unclear* code applied to opinions in which a positive/negative assessment could not be made. Whereas the theme coding allows assessment of the topics raised by survey participants, the positive/negative coding supplies a rough accounting of whether the survey comments were praising or criticizing LabVIEW. However, care must be used in interpreting the positive/negative counts. The positive/negative codes cannot reflect the degree of enthusiasm. So, both the assertions 'LabVIEW is outstanding' and 'My opinion is favorable' would receive the same code.

After the coding step was completed, the final step was to calculate three tallies for each theme:

- *Positive tally*. The number of respondents who gave a positive opinion falling within the theme.
- *Negative tally*. The number of respondents who gave a negative opinion falling within the theme.
- *Other tally*. The number of respondents who gave opinions that were conditional or unclear.

Note that these measures count the number of respondents expressing positive (or negative) opinions in a theme, not the total number of distinct opinions. If one respondent made multiple statements falling into the same theme, this contributed a count of one to the tally for that theme. Also, intersection is possible between the three tallies for a given theme; some respondents expressed both positive and negative opinions within the same theme, for example, thus contributing to both tallies.

Both authors participated in the qualitative analysis, and the results were assessed to determine the level of inter-rater reliability. After an initial set of theme definitions was suggested based on a reading of the data, a random sample of 40 responses (roughly 17% of the total coded comments) was selected from the total 227 responses. Each of the authors coded this sample independently, after which the two sets of codes were compared. In cases where differences occurred, either the theme definition was altered or a change was agreed for one of the codes. Thus, the two authors were able to reach close understandings of the theme definitions. The first rater then completed coding all 227 responses, in accordance with the agreed upon theme definitions. Finally, the second rater performed a blind coding of a further random sample of 20% of the total comments. The level of inter-rater reliability for that sample was 92%. Given this high level of agreement, the results reported here are based on the coding made by the first author.

4. Quantitative Analysis

4.1. Sample Characteristics

The first section of the survey asked respondents to describe themselves. The resulting profile of the survey sample is illustrated in Figures 1–3.

Figure 1 breaks down the types of programmers based on the data collected in Question 2 of the survey. This data allows respondents to be classified as professional programmers, end users (people who are not computer professionals, yet whose jobs involve some programming) and computer academics. Almost all (91%) of the respondents were professional programmers or end users. Since there were very few computer science academics, statistical analysis based on programmer type was limited to looking for differences between the professional programmers and end users.

Figure 2 shows the reported amount of overall programming experience and LabVIEW programming experience. The general profile of experience indicates that respondents had substantive LabVIEW experience; 78% of respondents reported that they had at least written medium-sized programs in LabVIEW. Although the

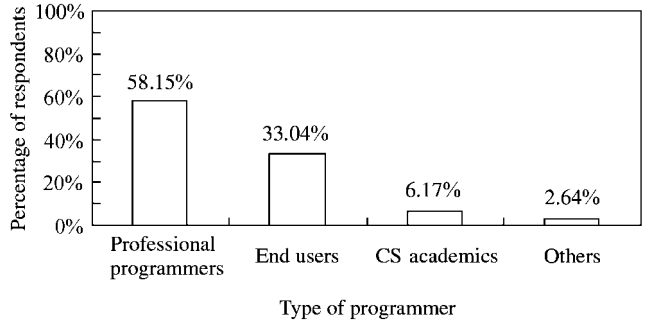


Figure 1. Type of programming experience reported in Question 2

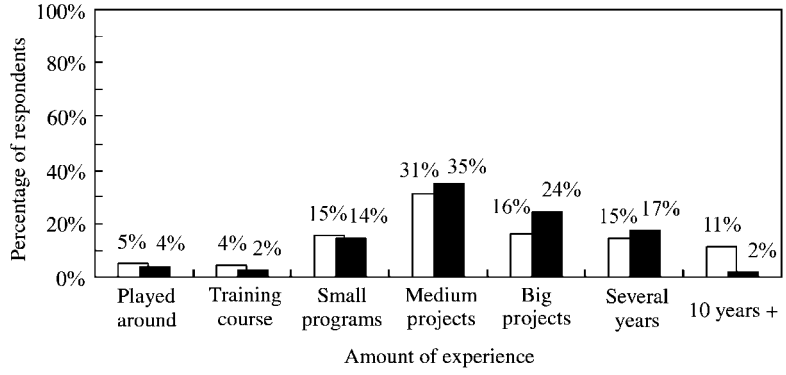


Figure 2. Amount of programming experience reported in Question 3: (□) general programming; (■) lab view programming

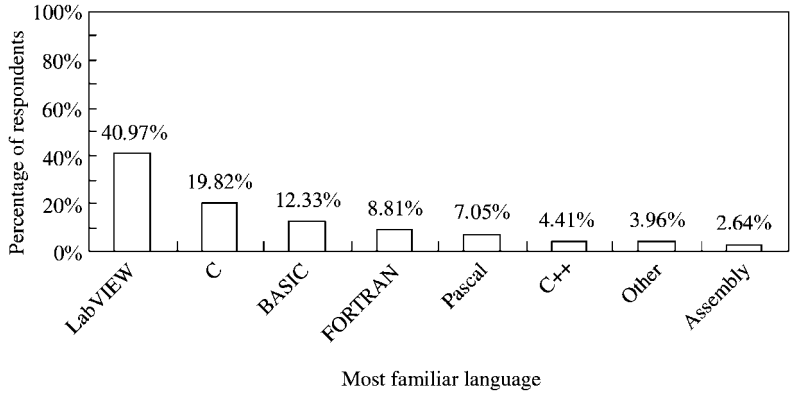


Figure 3. Most familiar languages reported in Question 4

questionnaire design intended that respondents would interpret 'overall programming' as meaning total experience including both LabVIEW and all other languages, 66 respondents (29%) reported that they had more LabVIEW experience than overall experience. It is therefore clear that respondents interpreted this question in different ways. Special care was taken in further analysis that relies on this distinction.

Figure 3 shows the languages with which respondents reported having the most experience. Respondents often listed multiple languages; the first language listed was the one used when it was necessary to attribute experience to one specific language. A largest group (41%) had used LabVIEW more than any other language. The majority of the remainder listed experience with C, BASIC, Pascal and/or FORTRAN.

4.2. Hypothesis Tests

Hypothesis 1 stated that respondents would rate the value of LabVIEW's reusable libraries higher than LabVIEW's visualness. This hypothesis was not supported by the ratings from Question 6. In fact, the respondents valued LabVIEW's visualness (a mean* of 5.43) slightly more than LabVIEW's reusable libraries (5.43 versus 5.29, a mean difference of 0.14, significant by a paired samples t-test at $t(215) = 2.16$, $p < 0.05$).

Hypothesis 2 stated that respondents would rate G more poorly than textual programming in its support for conditional logic. Based on the ratings collected in Question 9, Hypothesis 2 was not upheld. The mean rating given to this factor for textual languages is higher than that given to LabVIEW, but a paired samples t-test showed that this difference is not statistically significant (4.83 versus 4.70, a mean difference of 0.13, $t(204) = 1.05$, $p = 0.29$)†. However, when restricting the focus to

*The means for Hypothesis 1 differ slightly from the means in Table 1 due to the intricacies of performing a paired samples t-test. If a respondent failed to give a rating to either feature, he/she could not be included in the t-test.

†This analysis includes all respondents who were able to name a textual language used in their comparison. As in all analysis based on Question 9, respondents who were guessing about textual programming are not included. See Section 4.4 for further information.

respondents reporting greater overall experience with programming than with LabVIEW programming (Question 3), one finds a significant difference. These respondents assessed conditional logic at 5.14 for text versus 4.62 for LabVIEW (a mean difference of 0.52, significant at $t(70) = 2.82$, $p < 0.01$). Similarly, when looking at respondents who were more experienced in a language other than LabVIEW (Question 4), one finds a consistent significant difference (5.02 versus 4.57, a mean difference of 0.46, significant at $t(124) = 2.85$, $p < 0.01$). Perhaps programmers perceive more problems with LabVIEW's support for conditional logic as their experience with other languages gives them more alternatives to compare to LabVIEW.

4.3. Ratings of LabVIEW Features

In Question 6, respondents rated six of LabVIEW's features according to how much each is an advantage or disadvantage of using LabVIEW. The mean ratings, presented in Table 1, ranged from 4.62 (Customer Service for LabVIEW) to 5.41 (LabVIEW's Graphical Language (G)). Overall, the respondents had a positive assessment of all of the LabVIEW features; the rating scale allowed ratings between one and six, so all mean ratings are above the midpoint (3.5) on the scale. Analyses of variance (ANOVAs) were done to check for differences due to type of programmer (professional programmers versus end users) and language experience (LabVIEW versus all other languages given in Question 4). Neither factor had a significant effect on the Question 6 ratings.

Looking at the relative perceived importance of the LabVIEW features shows a particular preference for LabVIEW's visualness. As reported above, respondents valued LabVIEW's visualness significantly more than LabVIEW's reusable libraries. In fact, respondents rated LabVIEW's visualness significantly higher than they rated each of the five other features (as shown in Table 2).

Another interesting comparison is between the ratings for LabVIEW's visualness and LabVIEW's basis in the dataflow paradigm. In both the quantitative and qualitative analyses, a distinction is drawn between these two concepts. However, an obvious question is whether the survey respondents understood the distinction. Respondents rated the value of LabVIEW's dataflow paradigm significantly lower than LabVIEW's visuals. Yet, these two variables are more strongly correlated than any pair of LabVIEW features; visuals and dataflow are correlated at $r = 0.66$, $p < 0.001$, while the other correlations between Question 6 variables range from $r = 0.23$ to 0.46). This suggests the idea that, although respondents considered visualness to be the most important

Table 1. Means and standard deviations of the ratings reported in Question 6

LabVIEW Feature	Number of responses	Mean	S.D.
LabVIEW's Graphical Language (G)	223	5.41	0.99
LabVIEW Toolkits/Reusable VIs	219	5.28	0.85
LabVIEW's Support for 'Front Panel' Interfaces	223	5.20	0.98
G's Use of Dataflow Programming	219	5.02	1.03
National Instruments Hardware	201	4.95	1.04
Customer Service for LabVIEW	194	4.62	1.17

Table 2. Paired samples *t*-tests comparing LabVIEW's graphical language to all other LabVIEW features rated in Question 6

LabVIEW's Graphical Language (G) compared to all other LabVIEW features	Means	*t*	*df*	*p*
LabVIEW's Graphical Language (G) LabVIEW Toolkits/Reusable VIs	5.43 5.29	2.16	215	< 0.05
LabVIEW's Graphical Language (G) LabVIEW's Support for 'Front Panel' Interfaces	5.40 5.20	2.92	218	< 0.01
LabVIEW's Graphical Language (G) G's Use of Dataflow Programming	5.41 5.03	6.61	216	< 0.001
LabVIEW's Graphical Language (G) National Instruments Hardware	5.43 4.95	6.04	197	< 0.001
LabVIEW's Graphical Language (G) Customer Service for LabVIEW	5.41 4.61	8.48	189	< 0.001

aspect of LabVIEW, they thought that the visualness and dataflow are closely related concepts in the context of LabVIEW. For additional discussion about dataflow and about the respondents' understanding of dataflow, see Section 6.3.

4.4. Ratings of Graphics versus Text

In Question 9, respondents compared visual programming (as provided by G) to textual programming (as provided by a language of their choice) along seven different dimensions: power, writeability, readability, modifiability, enjoyability, whether the language adequately supports repetitive logic and whether the language adequately supports conditional logic. Respondents were asked to rate textual programming, even if they had no experience using a textual language; these respondents were instructed to enter 'guess' in the field where they identified a textual language. In a previous paper [5], the opinions of LabVIEW users who had no experience with text programming were compared to those of professional programmers who had no experience with visual programming; that study emphasized preconceptions about textual and visual representations, as investigated earlier by Blackwell [4]. The present study emphasizes informed opinions; the following analysis, therefore, excludes the ratings of the 21 respondents who said that they were guessing about their textual ratings.

As in Question 6, the average response was positive about G. As a general assessment, the mean total of ratings for G is 34.7 out of a possible 42, while the mean total for text is significantly lower at 28.7 (a paired samples *t*-test yields $t(199) = 9.03$, $p < 0.001$). On the six-point scale, the mean totals are equivalent to a rating of 4.95 for LabVIEW and 4.10 for text. Table 3 describes each rating dimension.

Analysis of Question 9 involves contrasting the dimensions to investigate where the respondents think that the strengths of visual programming truly lay. The comparisons involve not only the absolute ratings for each dimension, but also the difference between the corresponding ratings each subject gave to G and the textual language. Table 4 shows the mean of the difference scores for each dimension, which are calculated by subtracting the textual rating from the rating for G; positive means indicate higher scores, on average, for G than for text.

Table 3. Means and standard deviations of the ratings reported in Question 9

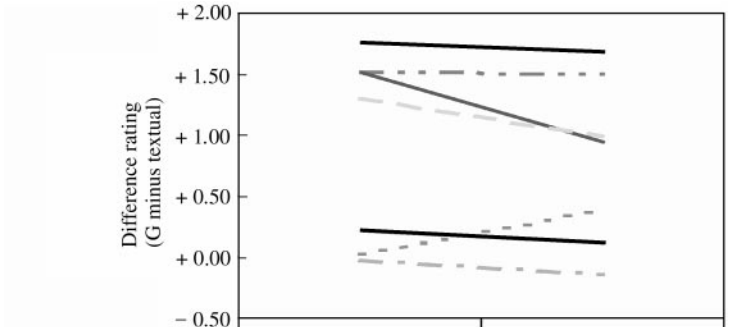
Rating Dimension	Responses (G)	Mean (G)	S.D. (G)	Responses (textual)	Mean (textual)	S.D. (textual)
Enjoyable	205	5.07	1.16	203	3.37	1.39
Easy to write	206	5.14	1.02	206	3.60	1.28
Easy to read	206	4.68	1.19	206	3.50	1.19
Easy to modify	206	4.81	1.25	206	3.67	1.35
Powerful	205	5.21	0.78	206	4.95	1.25
Easily supports repetitions	206	4.98	1.12	206	4.83	1.11
Easily supports conditions	205	4.70	1.22	206	4.83	1.18

Table 4. Means and standard deviations of the differences (G minus textual) between ratings reported in Question 9

Rating dimension	Number of responses	Mean (G − textual)	S.D.
Enjoyable	202	+ 1.72	+ 2.04
Easy to write	206	+ 1.54	+ 1.84
Easy to read	206	+ 1.19	+ 1.95
Easy to modify	206	+ 1.14	+ 2.11
Powerful	205	+ 0.27	+ 1.51
Easily supports repetitions	206	+ 0.15	+ 1.59
Easily supports conditions	205	− 0.13	+ 1.79

These ratings differences were affected by both programmer type and language experience. To look at programmer type, an ANOVA was done with two factors: programmer type (a between-subjects factor with the 70 professional programmers compared to the 114 end users) and the seven ratings differences (a within-subjects factor). The ANOVA showed a significant interaction between dimension and programmer type ($F(6, 1092) = 2.15, p < 0.05$). As illustrated in Figure 4, the interaction is due to the fact that the difference scores are not the same. Some are larger than others, and some are in different directions. For example, the professional programmers rated G relatively higher for readability than the end users, but the professional programmers rated G relatively lower for power than the end users. None of the individual difference scores were significantly different when comparing the professional programmers and the end users. One comparison was marginally significant. Compared to the end users, the professional programmers considered the advantage of G to be higher for readability (+ 1.51 versus + 0.95, a mean difference of + 0.56, marginally significant by an independent samples t-test at $t(188) = 1.90, p < 0.06$).

In the case of language experience, one finds a stronger effect. Here, the ANOVA consisted of language experience (a between-subjects factor with the 79 respondents who named LabVIEW in Question 4 as the language they had most experience using compared to the 121 remaining respondents) and the seven ratings differences (a within-subjects factor). As displayed in Figure 5, the ANOVA showed a main effect due to language experience; respondents who had mainly used LabVIEW rated G higher overall than all other respondents ($F(1, 198) = 29.58, p < 0.001$). Additionally, an



	Professional programmers	End users
Enjoyable	+ 1.76	+ 1.68
Easy to write	+ 1.51	+ 1.50
Easy to read	+ 1.51	+ 0.95
Easy to modify	+ 1.30	+ 0.98
Powerful	+ 0.01	+ 0.38
Repetitions	+ 0.21	+ 0.12
Conditions	− 0.03	− 0.14

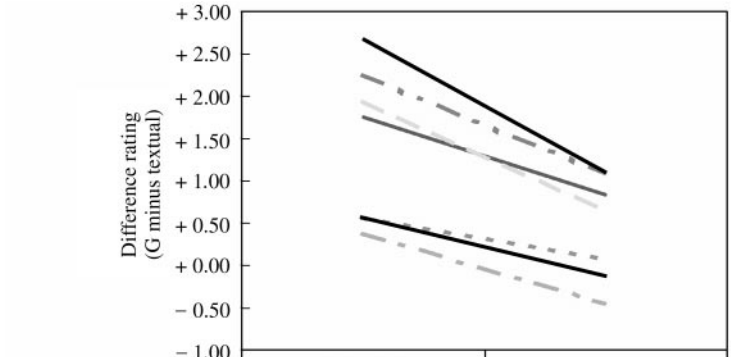
Type of programmer

Figure 4. Means of the ratings differences (G minus textual) for the professional programmers and the end users

interaction result between dimension and language experience reflects the fact that degree of difference was not constant for each dimension; the mean differences between the two groups ranged from + 0.50 for power up to + 1.57 for enjoyability. The interaction does not contradict the main effect, though, which clearly shows consistent higher ratings as a result of language experience; the respondents who had mainly used LabVIEW rated each dimension significantly higher than the rest of the respondents.

To glimpse into the conceptual structure of the respondents' ratings, factor analysis using principal components extraction and varimax rotation was applied to the raw visual and textual ratings. Because differences were found in the ANOVAs, factor analyses were done separately for the two groups based on type of programmer and also for the two groups based on language experience. The results were nearly identical for each group. Therefore, another factor analysis was done on the data as a whole; the data reported here is for this combined analysis.

Factor analysis resulted in three main factors, which cumulatively accounted for 63% of the variance. Separately, Factor 1 accounted for 25% of the variance, Factor 2 accounted for 22% and Factor 3 accounted for 16%. Table 5 shows the factor loadings for the three factors; in this table, loadings less than ± 0.25 have been set to zero. Factor 1 represents a visual factor, as all ratings for LabVIEW load highly on this factor, while all the ratings for text have very low loadings. Factors 2 and 3 represent textual factors. In Factor 2, respondents grouped together 'usability' aspects of textual programming, whereas in Factor 3, they grouped 'computational' aspects. The usability factor



	LabVIEW	Not LabVIEW
Enjoyable	+ 2.67	+ 1.11
Easy to write	+ 2.26	+ 1.08
Easy to read	+ 1.75	+ 0.83
Easy to modify	+ 1.93	+ 0.64
Powerful	+ 0.57	+ 0.07
Repetitions	+ 0.57	− 0.12
Conditions	+ 0.38	− 0.46

Most familiar language

Figure 5. Means of the ratings differences (G minus textual) for the respondents who named LabVIEW as the language they were most experienced in using and the remainder of respondents

Table 5. Factor loadings for the three factors resulting from the factor analysis on the ratings reported in Question 9. Loadings less than $\pm$ 0.25 have been set to 0.00

Rating dimension	Factor 1	Factor 2	Factor 3
LabVIEW			
Enjoyable	0.74	0.00	0.00
Easy to write	0.75	0.00	0.00
Easy to read	0.67	− 0.26	0.00
Easy to modify	0.71	0.00	0.00
Powerful	0.63	0.00	0.00
Easily supports repetitions	0.70	0.00	0.00
Easily supports conditions	0.70	0.00	0.00
Textual			
Enjoyable	0.00	0.74	0.31
Easy to write	0.00	0.85	0.00
Easy to read	0.00	0.87	0.00
Easy to modify	0.00	0.79	0.28
Powerful	0.00	0.00	0.74
Easily supports repetitions	0.00	0.30	0.85
Easily supports conditions	0.00	0.25	0.85

was a cluster composed of the textual ratings for enjoyability, writeability, readability and modifiability. The computational factor was a cluster composed of the textual ratings for power and support for repetition and conditional logic.

Considering the question of why the factor analysis revealed two textual factors and only one visual factor, one can, at best, speculate. Perhaps the rating dimensions are somehow more integrated in G. Alternatively, perhaps these respondents are somehow less able to distinguish between the concepts within G code. For example, maybe the respondents were more experienced with textual programming languages and, therefore, more discerning in their textual ratings. This last idea, however, is not consistent with the information from the individual factor analyses. The factor analyses for three of the four groups (the professional programmers, end users and the respondents who listed LabVIEW as the language they were most experienced using) resulted in the same factors as in the overall analysis, including very similar factor loadings and percentages of variance. Thus, even the respondents with less experience in textual programming (i.e. the respondents who listed LabVIEW as their main language) were equally discerning in their textual ratings. Indeed, the opposite effect seems more likely, namely that textual programming experience enabled discerning ratings of G. This possibility is consistent with the separate factor analysis of the respondents who listed a language other than LabVIEW as their main language. The vast majority of the languages that they did list were textual; all but eight listed C, Basic, FORTRAN, C++ or assembly language. This factor analysis resulted in the two textual factors (with similar factor loadings and percentages of variances). However, instead of one visual factor, these respondents had two visual factors, the first grouping the usability aspects of G and accounting for 19% of the variance and the second grouping the computational aspects of G and accounting for 14% of the variance).

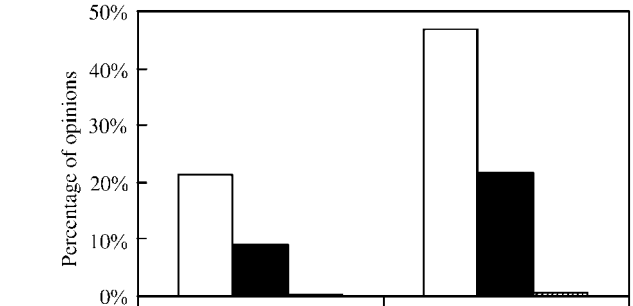
Overall, respondents saw the benefits of G in its usability, not its computational aspects. Taking the average scores of the usability group, a paired samples *t*-test between the visual and textual ratings is significant; respondents rated G higher than text (4.93 for G versus 3.53 for text, a mean difference of 1.39, significant at $t(205) = 11.86$, $p < 0.001$). In contrast, respondents did not rate G higher than text for the computational aspects (4.96 versus 4.87, a mean difference of 0.09, $t(205) = 1.03$, $p = 0.31$).

4.5. Quantitative Analysis of Open-Format Responses

Coding the open format responses yielded a total of 1746 tallied opinions from the 227 survey responses. This total represents enthusiastic participation from survey respondents, who contributed an average of 7.7 tallied opinions apiece. Figure 6 shows the distribution of the open-format responses.

Just as in Questions 6 and 9, the respondents were positive in their assessment of LabVIEW, with 68% of statements coded as positive. In general, respondents who had primarily used LabVIEW made a greater number of positive statements; respondents who named LabVIEW as the language that they had most experience using made 5.89 positive statements on average, while those who named other languages made only 4.80 (a mean difference of 1.09, significant at $t(207.75) = 3.30$, $p < 0.001$).

As for the attention given to the visual aspects of LabVIEW, 537 (31%) of the opinions were specifically related to the visual language G. Thus, respondents gave twice as many comments about non-visual issues. Different possibilities could explain why



	Visual themes	Non-visual themes
☐ Positive	21.36%	46.85%
■ Negative	8.93%	21.53%
▨ Other	0.46%	0.86%

Figure 6. Comparison of the total tallies for the visual versus non-visual comments made in the LabVIEW survey expressed as a percentage of the total tallied (1746) opinions

there are so many more non-visual comments than visual ones. As noted previously, a conservative coding policy was necessarily enforced. A respondent commenting about a productivity increase may believe that the productivity increase stems from G's visual syntax; however, if that respondent did not explicitly mention G's visualness, his or her comment would be coded as non-visual. On the other hand, the wide-ranging non-visual comments indicate the panoply of factors that combine to form the entire programming language and environment. In LabVIEW's case, several non-visual features provide candidate explanations accounting for the bulk of LabVIEW's impact on programmers. These features are described in the analysis of the non-visual themes (Section 6).

5. Qualitative Analysis: Visual Themes in Open-Format Responses

The survey yielded 537 tallied opinions about LabVIEW's visual representation, which were coded using the 16 visual themes. Table 6 defines the visual themes. The remainder of this section summarizes the themes that received the largest amount of attention from the survey participants.

Care must be taken in interpreting data collected using open-format questionnaires. Simply because a respondent does not give an opinion on a particular theme, one cannot conclude that the respondent does not have an opinion on that theme or think that that theme is an important issue. Bearing this caveat in mind, one can measure the attention that the LabVIEW respondents gave to different themes by looking at the percentage of respondents who chose to comment on a theme. Figure 7 gives these percentages for the visual themes. Figures 8 and 9 present the percentage of respondents making positive/negative tallies for each of the 16 visual themes. Figures 8 and 9 omit the percentages for conditional and unclear codes in the LabVIEW survey, as the total

Table 6. Definitions of the visual themes

General impact	Comments about ease of use; comments about whether ease of use renders visual programming more accessible to inexperienced users (e.g. students, end users, children) who might otherwise find textual programming to be too difficult
Learnability	Comments about whether a visual representation is easily learned; comments about whether a visual representation is intuitive, natural or obvious
Productivity	Comments attributing changes in productivity to a visual representation
Readability	Comments about how a visual representation fares in representing different aspects of program structure and behavior
Documentation effects	Comments about how a visual representation affects the need for additional program documentation
Syntax reduction	Comments about how a visual representation affects programming syntax
Modularity	Comments about whether a visual representation supports modular programming
Retention of text	Comments about the role that text plays in visual programming; comments about how text can be most beneficially combined with visual representations
Power	Comments about language level (the level of abstraction provided by a visual representation); comments about the scalability of a visual representation
Human visual capabilities	Comments about human perception of visual versus textual representations
Mental models	Comments about whether humans 'think in pictures'; comments that a visual representation is closer to human mental concepts than a textual representation
Preference	Comments that a visual representation is either fun (enjoyable, etc.) or frustrating (tedious, etc.)
Applying real-world experience	Comments about the role of direct manipulation in visual programming; comments about the role of metaphor in visual programming
Making the abstract concrete	Comments about the abstractness or concreteness of visual representation in which a respondent refers to the abstract nature of thought
Natural language	Comments comparing a visual representation to pictographic natural languages; comments that a visual representation is good internationally, because of independence from natural language
Miscellaneous	Comments that do not fit into any other visual theme

conditional and unclear codes over the entire visual data set was quite small (only 1.49% of the 537 opinions).

As an aid to placing the LabVIEW respondents' comments into a broader perspective, survey comments are linked to Green's cognitive dimensions throughout the description of all of the survey themes (both in Sections 5 and 6). Green's cognitive

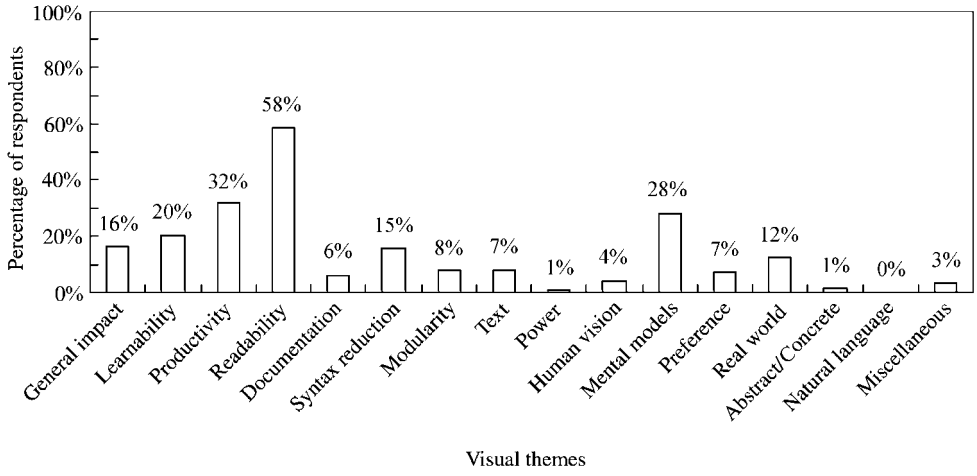


Figure 7. The percentage of LabVIEW respondents expressing an opinion in a visual theme

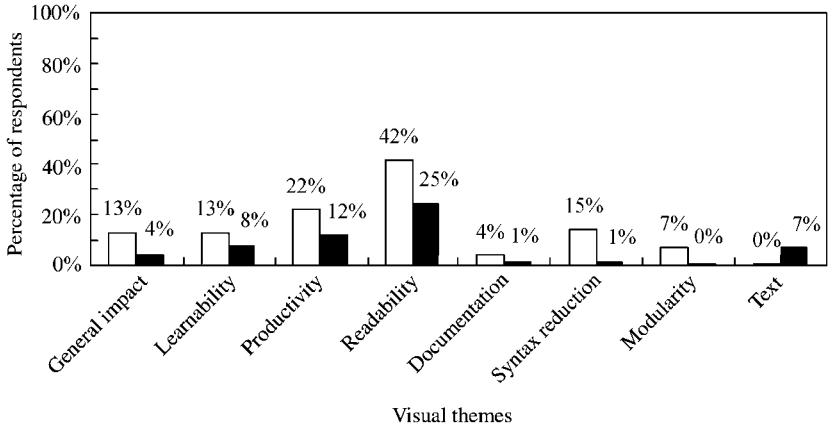


Figure 8. The percentage of LabVIEW respondents expressing a positive or negative opinion in eight of the 16 visual themes (also see Figure 9): (□) positive; (■) negative

dimension framework provides a vocabulary for describing the cognitive aspects of notations, including programming languages [1]. Linking the LabVIEW comments to the cognitive dimensions also provides a comparison between the opinions gathered in the LabVIEW survey and Green and Petre's informal evaluation of LabVIEW [2]. Definitions of the cognitive dimensions are supplied as they are introduced into the discussion. As an additional memory aid, Table 7 supplies abbreviated definitions of all of the cognitive dimensions mentioned in this paper; these definitions were taken from Green and Petre's definitions [2]. No attempt was made to discuss exhaustively how all of the cognitive dimensions apply; 11 of the 13 cognitive dimensions given in [2] were chosen as most relevant to the discussion of LabVIEW.

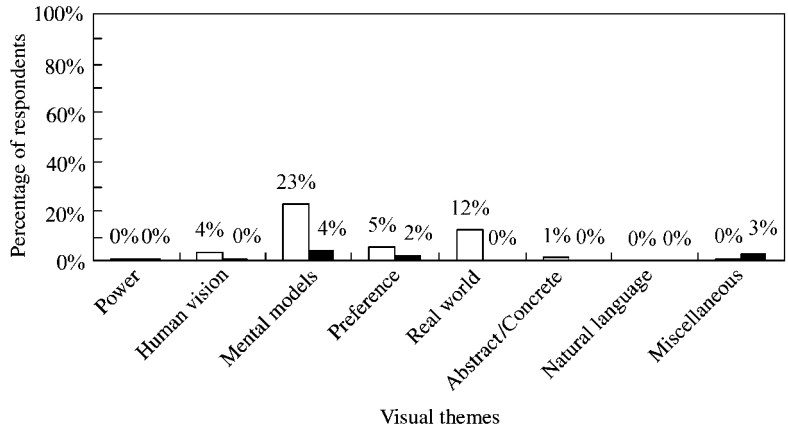


Figure 9. The percentage of LabVIEW respondents expressing a positive or negative opinion in eight of the 16 visual themes (also see Figure 8): (□) positive; (■) negative

Table 7. Glossary of the cognitive dimensions cited in this paper. These definitions were taken from Green and Petre's definitions [2]

Abstraction gradient	How many types of abstractions does the language demand and/or allow?
Closeness of mapping	How closely does the program world match the problem world?
Diffuseness	How much screen space (symbols/graphic entities) does the code require?
Error-proneness	Does the notation induce 'careless mistakes'?
Hard mental operations	Is the code hard to follow without the aid of fingers or penciled annotation?
Hidden dependencies	Are all dependencies explicitly indicated in both directions?
Premature commitment	Do programmers have to make decisions before having all relevant information?
Progressive evaluation	Can partially complete code be executed to obtain intermediate feedback?
Secondary notation	Can programmers effectively use layout to convey meaning informally?
Viscosity	How much effort is required to make a single change?
Visibility	Are all parts of the code simultaneously visible? Can different parts be juxtaposed side-by-side at will?

5.1. Visual Theme: Readability

Readability received more attention in the LabVIEW survey than any other visual theme (58% of the respondents). Whereas the LabVIEW respondents regularly commented on whether G code is easy to read, they did not supply enough details to allow a more fine-grained categorization. Fortunately, some repeated ideas suggest avenues for further exploration. One of the more striking subgroups to arise from this theme was the

frequently stated opinion that G code provides an overview perspective to the programmer. LabVIEW respondents referred to this as a fast and easy grasp of 'the big picture', 'the Whole Picture', 'the Gestalt view', etc. (respondents might have also been referring to this idea when they used phrases like 'G clarifies the structure'). Moreover, they suggested that this overview perspective is well integrated with the details of the code: for example, 'The little boxes help you compartmentalize a design and understand both the big picture and the details' and 'It's easier to view the global scope of the work while working on details.'

It is unclear how the comments about an overview perspective fit into the cognitive dimensions framework. Perhaps the respondents were commenting about visibility, the cognitive dimension that defines how readily program information is available to the programmer. Alternatively, perhaps the respondents were describing a consequence of a good level of hidden dependencies. Hidden dependencies is the cognitive dimension regarding whether a notation overtly indicates every dependency in both directions and whether this indication is accomplished perceptually or symbolically. Indeed, in other comments, LabVIEW respondents often liked the way that G makes 'connectivity' and data dependencies easy to follow.

Yet a price of making more of a program's relations visible appears to be paid in the areas of hard mental operations, secondary notation and premature commitment. Such tradeoffs between cognitive dimensions are unavoidable according to Green. He gives as an analogy the interaction between temperature, volume and pressure as characteristics used in describing a physical body of matter; changing the pressure of a body of gas cannot be done independently of its other dimensions, for example [1]. Hard mental operations is the cognitive dimension concerning whether problematic notational constructs force a programmer to resort to penciled annotations or fingers tracing over code displayed on a screen in order to keep track of what is happening. In LabVIEW, all local dependencies are displayed (i.e. a low degree of hidden dependencies); a possible consequence may be an increased number of hard mental operations as a programmer tries to trace individual execution paths of a program. Secondary notation refers to the use of layout cues to encode meaning outside of the formal semantics of the notation. Again, LabVIEW's explicit representation of dependencies coupled with the two-dimensional nature of LabVIEW code, may make achieving good secondary notation difficult; ironically, the value of good secondary notation is likely to be higher in LabVIEW due to the same influences (no hidden dependencies and hard mental operations). Finally, premature commitment, in this case, refers to decisions required in the layout of LabVIEW code in which a programmer is forced to make initial layout decisions before the shape of the program is fully known. Due to LabVIEW's high viscosity (as noted in *Productivity*) and large number of visible entities, mistakes in premature commitment in layout are likely to be problematic.

Green and Petre noted problems in hard mental operations, secondary notation and premature commitment in their assessment of LabVIEW. In the LabVIEW survey, respondents made observations consistent with Green and Petre's; at least 20 respondents reported upon the increased complexity of achieving good readability in G code. These respondents explained that, in G, it is very easy to create messy, cluttered, hard to read 'bird's nests'/'spaghetti code'. Thus, LabVIEW programmers must allocate time for organizing code layout. One LabVIEW programmer suggested that there might be a benefit to working on layout: 'Have noticed that when making the diagram neater,

the logic seems to clean itself up at the same time.' Others were not so sanguine: for example, 'This detracts from actual work getting done.' In fact, one respondent solved the problem by hiring extra help: 'Recently, I hired a real nitpicker to clean up my diagrams while they are in progress. That saves me a lot of time and my customers get even neater diagrams than I would have done.'

As for comments about specific visual constructs in G, respondents supplied some details about constructs for loops, conditional statements and sequence structures. Several commented that sequence structures tend to be cryptic: for example, 'Programming sequences is trivial in a procedural language. It is arcane in LabVIEW. Sequence structures are obscure. Adding artificial data dependencies is also obscure, and … spreading a sequence all over a large [area] that is difficult to read, or else over multiple pages, which is even more difficult to read.' While a few respondents seemed pleased about G's loops and conditional constructs (e.g. 'it's easier to see the structure components [in the diagram] (for example, a loop) than in a text language'), others supplied detailed criticisms. Some noted difficulties in nesting constructs: for example, '… programming some things (like a series of logic tests, or nested 'if−thens') seems clumsy in LabVIEW'. A few did not like the fact that G splits up different branches of its conditional statements (if statements and case statements) into different screens: for example, 'you never see all components of a loop or a case structure'.

This point about LabVIEW's conditional constructs falls into the cognitive dimension visibility; specifically, it is an example of poor local visibility. Green and Petre made the same assessment when they wrote, 'The demand placed on working memory by the 'invisible arms' is just too much' [2, p. 164]. Additionally, respondents noted another characteristic of LabVIEW code that allows poor local visibility. LabVIEW actually adds a third dimension to program layout, in that program elements can be placed one on top of the other. As one respondent noted, this fact means that code can be obscured from view: 'Programming in G requires more organizing; cluttered diagrams are easy to create and very difficult to read (not just to follow, but just to see where everything is!). Even cluttered textual languages are easy to read (nothing is ever covered up on the screen!).' The obscured code complicates the readability not only of onscreen code, but also of printed code; two respondents mentioned the problem in the context of attempts to print LabVIEW code.

Finally, another subgroup of the readability comments concerns the information density of visual code (i.e. the amount of space it takes to express an amount of information). This issue is called diffuseness in Green's cognitive dimensions. The information density of a notation can be affected by different factors, including both the space required for individual constructs within a notation and how much information a notation makes explicit. In Green's definition, he emphasizes that diffuseness is a measure that is independent of whether the notation provides a lot of high-level primitives [2]; in such a language, fewer lexemes would be required to express an amount of information, thus allowing the language to be compact in a different way. As an example of differences between visual and textual code, encoding arithmetic expressions using visual representation can be much less compact than using textual representation. As a result, the code may become less readable (which is perhaps why some LabVIEW respondents commented on the need for textual arithmetic representation in the theme *Retention of Text*).

Aside from comments about mathematical formulas, LabVIEW respondents were divided on the issue of how G code fits within the screen real estate. Some noted that G code is generally larger than textual code: for example, 'Large programs tend to be much larger than the screen'; 'Some basic constructs are long winded … when they can be more easily handled in a language such as C, e.g. for next loops.'; and 'The writing of some algorithms in G takes much more place than its equivalent in C (bubble sort algorithm, for example).' Others felt that they could see more of an entire program on the screen at one time; these comments might be related to the idea that G successfully integrates a global view with the details of the program. For example, respondents said, 'Larger sections of code on the screen at one time allow better visualization of what you are doing' and 'You can visualize most of the program directly on one page. With text programs you have to roll back and forth through many pages of text.' Taken together, the comments are not a strong indication of whether LabVIEW's diffuseness characteristics are a help or hindrance to the programmer. Green and Petre felt that the diffuseness of LabVIEW was less of a hindrance than they had originally anticipated.

In considering other possible interpretations for some of the phrases coded into this theme, one finds fuzzy boundaries between *Readability* and a few of the other themes. For example, in cases where respondents indicated that G code is visually compact, the respondents might actually have been thinking about the fact that an icon can be used to represent an entire function. Such an opinion focuses on the fact that G provides a mechanism for modularity and would, therefore, be appropriate for either the visual or non-visual theme concerning modularity. Also, when respondents made comments like 'you can actually see the code', they may have had opinions about how similar G code is to their own mental models. However, unless respondents actually gave details about their own mental processes, such comments were placed here, as opposed to *Mental Models* (see Section 5.3). Finally, the comments about G's sequence structure (and, to a lesser extent, about G's other control constructs) illustrate the possible influence of the dataflow paradigm, a topic that is considered in the non-visual themes (Section 6). LabVIEW uses the dataflow paradigm as the basis for G, including versions of structured primitives familiar to programmers of imperative programming languages. When investigating whether any of G's structures is awkward, one should consider two possibilities: that the given structure may be poorly designed or that any attempt to integrate the structured concept into the dataflow paradigm may suffer from awkwardness. Green and Petre noted the need for more investigation into possible inherent characteristics of the dataflow model when they wrote, 'Our impression is that [problems with representation of control constructs] remains a problem in general with the dataflow model' [2, p. 167].

5.2. Visual Theme: Productivity

The productivity implications of LabVIEW's visualness attracted a fair amount of attention; with 32% of the respondents commenting in this theme, *Productivity* placed second only to *Readability*. There were a few comments about overall productivity gain and about the coding phase of development. More often, the LabVIEW respondents linked visual representation to benefits in testing and debugging. Incidentally, the coding of these debugging comments involves a distinction between visual representation and language responsiveness. The LabVIEW environment provides an animated debugger,

which makes it appear that data is flowing on a program's wires. In this survey's coding scheme, the responsiveness of the debugger is not coded as a visual aspect of LabVIEW, under the assumption that textual representations can be designed with an equal level of debugging responsiveness. One can hypothesize that the combination of a particular visual representation and responsiveness technique might produce a significant interaction; further research would be required to make that assessment.

In contrast to claims of faster and easier debugging, many *Productivity* statements were negative (12%). Respondents named one main problem leading to time delays in both code creation and modification. Namely, the layout of the code was repeatedly described as more time consuming: for example, 'LabVIEW requires that one spend perhaps an inordinate amount of time literally 'laying out' code to avoid a rat's nest of wires, even when the algorithm is well-thought out and clean. Again, difficult to modify later. I can easily add new code to a C/C++ function without having to worry if it will 'fit' and my editor (Emacs) takes care of the formatting leaving me free to concentrate better on the task at hand.'

Resistance to simple editing tasks (i.e. time costs) is described by the cognitive dimension called viscosity. The survey comments involving viscosity agree with Green and Petre's assessment that LabVIEW's viscosity is quite high and that, for VPLs in general, 'the role of the diagram editor is crucial' in determining whether a VPL's viscosity is at an acceptable level [2, p. 167]. It is likely that LabVIEW's viscosity is related to its low level of hidden dependencies. In similar, but not as frequent comments, respondents described other possible consequences of moving beyond line-based code: for example, 'Hard to cut and paste code'; 'It is difficult to modify these structures (e.g. remove a loop without removing the contents of it)'; 'Harder to temporarily comment code out in some cases.'; and 'Global search and replace is not an option of LabVIEW.'

5.3. Visual Theme: Mental Models

In the LabVIEW survey, *Mental Models* received the third highest attention (28%) of the visual themes. Of these comments, positive opinions outnumbered negative ones by approximately six to one. LabVIEW respondents repeated ideas that the designs they construct in their minds are in some sense pictorial: for example, 'I am more of a picture person when it comes to doing projects.' and 'I see pictures of data flow. That is how my mind works.' They also echoed ideas that G is closer in nature to their thoughts than are textual languages, thus narrowing or eliminating the mapping process which takes place when a programmer expresses thoughts within a programming language: for example '[G] makes it easier to assimilate and control the thought process. Don't have to clutter the brain with lines of code' and 'My feeling is that I translate LabVIEW-like representations of data manipulation into conventional code. With LV, I can skip that step, the wiring diagrams correspond pretty closely to the procedural description of the program I have in my head.' Of course, not everyone agreed that G fits so well with their mental models: for example, 'The thought process for program development seems much more complex with G' and 'LabVIEW is forcing visual which I am not inherently good at.'

The respondents' claims that LabVIEW's visuals allow them to express their thoughts more directly from their internal mental models certainly fall into the cognitive dimension closeness of mapping. Closeness of mapping is the dimension for delineating

how directly the concepts in the problem domain map onto the concepts supplied by the problem-solving notation. If these respondents are correct, they have fewer programming constructs to learn in order to solve their tasks.

In considering what prompted the LabVIEW respondents to claim that G is closer to their thoughts, note that at least three sources may have influenced their opinions. First, several may have been influenced by the predominance of diagrammatic representations for software design; some respondents explicitly broached the topic of familiar diagrammatic representations (see *Applying Real-World Experience*). However, it is noteworthy that programmers consider their internal representation to hold more in common with these diagrammatic representations than with the text that they eventually write. This introspective intuition has also been noted in a study by Petre and Blackwell [11]. Second, LabVIEW respondents may have actually been reacting to the use of the dataflow paradigm, as opposed to the G's visual representation of that paradigm. Third, the claimed narrowing of the semantic gap may be due to the issue of language level and not to the visualness of LabVIEW. These last two possibilities are explored further in Sections 6.5 (*Dataflow Paradigm*) and 6.6 (*Power*), respectively.

5.4. Visual Theme: Syntax Reduction

For some, visual programming provides freedom from the minutiae of programming language syntax. For example, some VPLs require fewer keywords and punctuation marks (braces, parentheses and semicolons, in particular). Similarly, some VPLs reduce the need to declare as many variables, via the use of graphical connections that can reduce the need to name intermediate values. Note, however, that VPLs do not eliminate syntax, but rather replace one set of rules with another, possibly more easily applied, set.

In the LabVIEW survey, 15% of the respondents broached the subject of syntax. They nearly unanimously applauded G's effect on programming syntax: for example, '[The] block diagram wiring convention is less prone to syntax and spelling errors' and 'Syntax errors are non-existent.' In Green and Petre's analysis of LabVIEW, they reasoned that VPLs, in general, reduce the syntax burden on the programmer and, consequently, that construction of programs is easier than in textual languages. In terms of the cognitive dimensions, Green classifies the effects of syntax into two dimensions: closeness of mapping and error-proneness. Whereas in *Mental Models*, respondents did not explain how LabVIEW was or was not close to their mental concepts, the comments in this theme provide a concrete example of closeness of mapping. In fact, syntax is just one of a large number of characteristics that affect the number of programming concepts required to achieve a solution to a problem; several other factors affecting LabVIEW's closeness of mapping will be discussed below. The error-proneness dimension concerns whether a notation's syntax rules tend to induce errors.

5.5. Visual Theme: Applying Real-World Experience

Twelve percent of the LabVIEW respondents praised LabVIEW's use of direct manipulation and/or familiar metaphors. The combination of the two was described as mirroring the real world: for example, 'G makes programming more real. In other

words, you feel like you are actually building something.' They noted that the direct manipulation in G exploits a highly developed human skill. When discussing metaphor, respondents noted LabVIEW's use of both flowchart notation and electronic circuit notation: for example, for electrical engineers and electronics students 'the wiring paradigm is much easier to grasp'; 'The visual layout is a good parallel of the way one thinks of signals flowing through circuits'; and 'G is written in the language of engineers: flow-charts.'

All of the comments in this theme fall into the cognitive dimension closeness of mapping. Direct manipulation provides a closer mapping between physical actions and effects, which is a benefit that applies to all users. Also, the more that LabVIEW's circuit metaphor fits a user's task, the more transparent the language becomes to the task.

5.6. Visual Theme: Retention of Text

Seven percent of the LabVIEW respondents described the role of text in G. This is a small percentage, yet in all but one case, respondents did so by noting that text is preferable to G's visual alternatives for writing mathematical formulas: for example, 'Complex formulas need to be done in C or equation nodes.' These respondents explain that using G's visual alternative for arithmetic expressions takes too long to write and that the resulting visual expressions are less readable and take up more screen space. As discussed in *Readability*, respondents mentioned problems with other aspects of G's visualness; however, in those cases, they did not explicitly recommend the use of text. As with similar comments in *Readability*, the comments in this theme involve two cognitive dimensions: diffuseness and hard mental operations.

Retention of Text illustrates a limitation of the positive/negative coding scheme. All of the comments preferring textual representation to G's visual representation for formula writing are coded as negative comments about LabVIEW's visualness. This coding does not reflect the additional viewpoints about the use of text in visual programming. Only one respondent decried the need for text: 'If you use a formula node, you might as well be using a textual language.' The remaining respondents in this theme seemed to accept the idea of a programming language incorporating both textual and visual elements.

6. Qualitative Analysis: Non-Visual Themes in Open-Format Responses

This section focuses on the 13 non-visual themes; Table 8 gives the definitions of these themes. Figure 10 gives the percentage of respondents who made a comment falling into each of the non-visual themes. Figures 11 and 12 present the percentage of respondents making positive/negative tallies for each non-visual theme. Figures 11 and 12 omit the percentages for the conditional and unclear codes in the LabVIEW survey, as the total conditional and unclear codes over the entire non-visual data accounted for only 1.24% of the tallied comments.

As in Section 5, this section limits its report to a subset of the findings in the non-visual themes. In this case, emphasis is given to non-visual attributes of LabVIEW that might explain a large part of LabVIEW's popularity. To summarize the other results in the non-visual themes, respondents were positive about the value of LabVIEW. In

Table 8. Definitions of the non-visual themes

General assessment	Comments about LabVIEW as a whole; comments about whether LabVIEW renders programming more accessible to inexperienced users (e.g. students, end users, children)
Learnability	Comments about whether LabVIEW is easily learned
Productivity	Comments about how LabVIEW affects productivity
Preference	Comments that LabVIEW is either fun (enjoyable, etc.) or frustrating (tedious, etc.)
Power	Comments about the language level of LabVIEW; comments about the scalability of LabVIEW
Application domain	Comments naming specific application domains that LabVIEW facilitates; Comments about whether LabVIEW is a general-purpose or domain-specific language
Reuse and modularity	Comments about all aspects of LabVIEW's facilities for reuse or modularity; comments about whether LabVIEW supports object-oriented programming
Front panel (GUI) setup	Comments about LabVIEW's features for building graphical user interfaces
Dataflow paradigm	Comments indicating an effect due to LabVIEW's use of the dataflow paradigm
Language features	Comments about LabVIEW language features in which visualness is not an issue
Sources of programmer support	Comments about a variety of sources for programmer support (e.g. LabVIEW manuals, phone support, training courses and the info-labview mailing list)
Implementation-specific remarks	Comments about LabVIEW's limitations due to implementation (as opposed to LabVIEW design decisions)
Miscellaneous	Comments that do not fit into any other non-visual theme

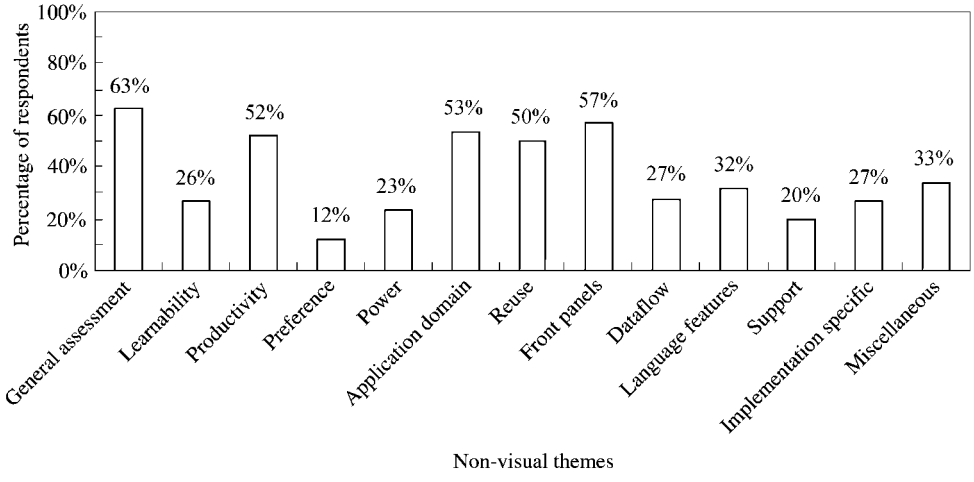


Figure 10. The percentage of LabVIEW respondents expressing an opinion in a non-visual theme

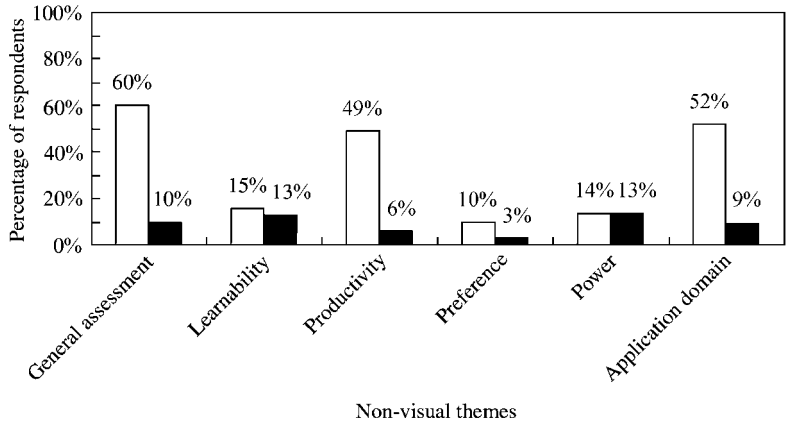


Figure 11. The percentage of LabVIEW respondents expressing a positive or negative opinion in six of the 13 non-visual themes (also see Figure 12): (□) positive; (■) negative

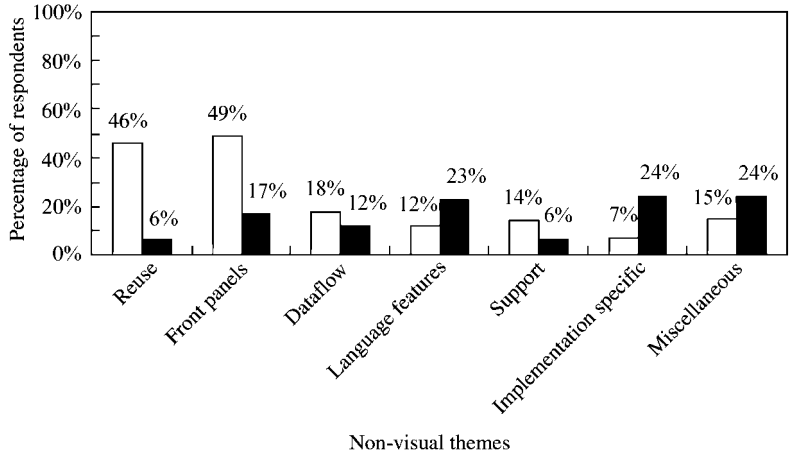


Figure 12. The percentage of LabVIEW respondents expressing a positive or negative opinion in seven of the 13 non-visual themes (also see Figure 11): (□) positive; (■) negative

General Assessment, respondents expressed enthusiasm for LabVIEW as a whole. So, while some respondents expressed relatively reserved opinions (by using phrases like 'my opinion is favorable'), many more gave stronger support (as evidenced by word choices like 'outstanding', 'impressive', 'excellent', 'wonderful', 'awesome', 'fantastic', 'superb', 'greatest' and 'revolutionary'). In *Productivity*, respondents reported benefits in overall development speed; for example, 'The speed of programming is amazing' and 'Very rapid development time for small to medium projects.' In fact, some respondents claimed impressive gains: for example, 'Comparisons I have personally taken part in have shown that the development time for LabVIEW is less than half that for C (to develop the same functionality)' and '[My company] has had excellent results, usually about 3X more productive than programming in C.'

Yet, not all of the non-visual themes were strongly positive. Three themes were negative in tone; negative comments outnumbered positive comments in each. *Language Features* included criticism of LabVIEW's lack of an event-handling primitive. Criticisms in *Implementation-Specific Remarks* centered on disk and runtime memory requirements and execution speed. No central topic dominated *Miscellaneous*, which included issues like platform independence and project management tools.

6.1. Non-Visual Theme: Reuse and Modularity

Fifty percent of the LabVIEW respondents made statements in this theme, thus placing it as one of the top non-visual themes in level of attention. Praise was given to LabVIEW's wealth of virtual instruments. VPLs are often designed to be domain-specific which, in practice, often means that they come complete with libraries of pre-constructed, reusable software routines. This is definitely the case in LabVIEW; in order to target the instrumentation market, National Instruments and associated vendors supply a wide variety of reusable routines including instrument drivers, data sampling routines and data analysis routines. In the survey responses, 59 respondents clearly noted the built-in VIs as advantageous: for example, 'I totally believe that LabVIEW is pioneering the way all programming will evolve to in the future. The ability it provides to create applications by mixing and matching pre-developed modules will help to make application development simple … ' and 'In LabVIEW you can write a program by seeing the function you need and placing it in your diagram.'

When broaching the subject of support for modularity, respondents often noted that LabVIEW fosters modular coding. A few respondents liked the fact that all VIs, whether a built-in VI or programmer-created one, are defined as independent components; these respondents linked this observation with the claims that LabVIEW makes unit testing easier. In contrast, a few respondents criticized the fact that LabVIEW does not support object-oriented programming: for example, 'A visual, OO system would be best' and 'Not enough features to support reuse, as in object-oriented languages …. Reuse is achieved through cut'n'paste rather than via abstraction.'

Relative to the cognitive dimensions framework, this theme touches upon three dimensions. All of the comments are relevant to the cognitive dimension abstraction gradient. Abstraction gradient concerns how much abstraction a notation supports and/or demands. Green and Petre define abstraction as 'a grouping of elements to be treated as one entity, whether just for convenience or to change the conceptual structure' [2, p. 144]. In LabVIEW, a basic level of abstraction is supplied by LabVIEW's support of control concepts (loops and conditionals). Additional support for abstraction comes via LabVIEW's subroutines, both the ones supplied to the programmer and the ones created by the programmer. On the negative side, the respondents lamenting LabVIEW's lack of object orientation feel that LabVIEW is not high enough on the abstraction gradient. Another relevant cognitive dimension is, arguably, progressive evaluation. Progressive evaluation is the idea that programming environments ought to support evaluation of partially completed programs, so that programmers can get intermediate feedback. While it is true that LabVIEW code cannot be executed until it is complete, the respondents liked the fact that all LabVIEW subroutines can be executed independently; this fact can be seen as a weak form of progressive evaluation. Finally, the cognitive dimension closeness of mapping plays an important role in this theme.

Certainly, the comments praising LabVIEW's code libraries indicate that LabVIEW is successfully increasing the closeness of mapping to some of its users' program domains. Compared to the relevant comments in *Mental Models*, the reuse comments illustrate a closeness of mapping that is independent of LabVIEW's visualness. Several other examples of closeness of mapping will appear in the non-visual themes.

6.2. Non-Visual Theme: Front Panel (GUI) Setup

LabVIEW divides a program into two elements: a block diagram and the user interface (which, in LabVIEW, is called the instrument panel or the front panel). Thus, LabVIEW is similar to languages like Visual Basic in that the language provides a division between the code that a programmer creates in dictating the behavior of a program and the work necessary to set up the program's graphical user interface. In LabVIEW, a programmer can select, place and customize all of the visible entities on the front panel without writing code. Furthermore, no code is needed to establish the connection between the input and output values seen on the front panel and the corresponding variables in a program's block diagram.

With 57% of the respondents, this theme ranked second in attention of the non-visual themes. Most of these comments were positive; several respondents were quite enthusiastic, describing the ease of front panel setup as LabVIEW's 'biggest advantage', its 'best feature', or its 'main power'. Some respondents liked the fact that LabVIEW 'divorces the user interface appearance from the program flow', thus allowing a logical organization of their software. Others described the components (e.g. switches, sliders and graphing options) available for building a GUI as easy to use and customize: for example, 'The front panel objects are great, you can just plop them down, resize, move them and so on' and 'Front panel indicator flexibility, such as for data passed to a graph: scaling, zooming in, transposing, etc. are all built into the front panel, so do not have to be coded.'

On the other hand, the negative comments falling into this theme indicate that LabVIEW users encounter difficulties when attempting to create more sophisticated user interfaces. They named several tasks that are difficult to achieve, such as radio buttons (i.e. a set of mutually exclusive choices) and dynamic front-panel modifications (for example, adding front panel controls programmatically). Similarly, respondents complained that LabVIEW does not supply the tools needed to create standard Windows or Apple interfaces (with menu bars and tool bars, etc.).

Additionally, several respondents tied their negative comments to the *Language Features* theme. In particular, they observed that LabVIEW does not support event-driven programming and that this predominantly affects their ability to implement user interfaces. LabVIEW does not provide a wait-for-event primitive. So, reacting to user actions on the front panel can require polling. Implementing flexible user interfaces is less straightforward as the programmer has to create more of the event-handling code. The following quotations illustrate these comments*: 'Some things are cumbersome,

*The quotations given here supply a fuller context to the phrases included in the *Front Panel* (*GUI*) *Setup* theme. In the actual coding of such statements, details about event handling are placed into *Language Features*, while comments about resulting impact on building user interfaces are included in *Front Panel* (*GUI*) *Setup*.

such as event handling (there is basically none, generally have to poll things, even front-panel buttons)'; 'LabVIEW does not handle event driven programming well. There are many instances in a program in which it needs to just 'sit there' and wait for the operator to do something to the user interface. LabVIEW is not as efficient at this as a language such as [Visual Basic]'; and '[LabVIEW] is great for making the computer do real-world tasks, but it can be difficult to build a good user interface. Basically, LabVIEW flows downstream, but a lot of times you want to give the user a 'cancel' button and go back upstream, and LabVIEW make this difficult. Similarly, it's not event-driven, so you have a hard time writing a program flexible enough to support all the events your users will dream up.'

This theme is closely related to the *Reuse and Modularity* theme. The rapid construction of simple GUIs amounts to reuse of software components. Indeed, some overlap is likely between the two themes; some respondents mentioning LabVIEW's VIs might have been implicitly referring to VIs for various data display options (e.g. a histogram) that direct how output is shown on the front panel. Only comments explicitly mentioning the user interface were included in this theme. In any event, for the users who are pleased with LabVIEW's GUI setup, it is a successful example of reuse and closeness of mapping. The task of implementing an interface becomes transparent; the programmer need not learn what Green and Petre call 'programming games' in order to achieve their goal [2].

6.3. Non-Visual Theme: Dataflow Paradigm

VPLs use different programming paradigms, just as do textual programming languages. A programming paradigm provides a set of concepts that define a virtual machine [12], whereas a graphical representation provides a mechanism for the programmer to specify computation in terms of said concepts. It seems reasonable to try to separate the effects of the representation of a notation from its paradigm. For example, one can design many visual and textual notations using the control flow paradigm; similarly, one can design visual and textual notations based on the object-oriented paradigm. If one were to test a visual, control-flow based language against a textual, object-oriented language, such an experiment would not allow clear conclusions about either the visual versus textual dimension or the paradigm dimension. In the dataflow paradigm, data travel in streams and pass through filter functions. A function can be executed when all of its operands are available; a low-level form of parallelism is achieved since the order of execution is defined solely in terms of data dependencies.

To attempt an initial guess as to whether LabVIEW's basis in the dataflow paradigm accounts for a substantial fraction of LabVIEW's effectiveness, consider the following. Some evidence suggests that, when solving problems, expert programmers are neither limited in their design behavior by their target programming language, nor by paradigm boundaries [13, 14]. On the other hand, Good argues for clearer separation between paradigm and representation in empirical studies and, specifically, for more investigation of control flow versus dataflow [15]. Indeed, some LabVIEW respondents, even ones who described themselves as experienced programmers, indicated that LabVIEW's dataflow paradigm may have an impact that is independent of LabVIEW's visual representation. If so, then LabVIEW's choice to use dataflow constitutes another example of closeness of mapping; the closer that dataflow paradigm fits the application

domain (e.g. the task of recording a stream of data being output from a hardware monitor), the fewer extra programming goals the programmer is required to solve.

Some comments in this theme lacked detail. Respondents stated that programming in dataflow is 'different'; for example, 'I am not well experienced in dataflow techniques, but I have seen LabVIEW programs written by those who have mastered dataflow concepts, and these have been very powerful and impressive programs.' Several respondents linked their comments about the dataflow paradigm to comments in the non-visual *Learnability* theme. These respondents said that the dataflow paradigm was the main adjustment required to learn LabVIEW: for example, 'Instead of procedure-based programming, it's data-dependency-based. This is a major change in philosophy, and was very difficult to comprehend for a die-hard C coder like myself. You have to get past the idea that 'A follows B', get away from recipe programming, and just look at the data dependencies.' and 'Having been programming in 'traditional' languages for 25+ years, I have a *VERY* hard time thinking 'dataflow'.'

In more detailed comments, respondents wrote about the parallelism that is intrinsic to the dataflow paradigm. Some noted that the 'inherent parallelism of data flow is very powerful.' Similarly, others stated that the dataflow paradigm '… allows for natural parallel execution of processes' and, thus, that 'the power of data flow programming will be in its use to program and to take advantage of the power of parallel processing computers.' Another aspect of specifying the order of operations is that some dependencies are easy to express (for example, 'Because of dataflow dependencies, synchronization of events becomes automatic.'). In contrast, lack of dependency can, perhaps, be easy to overlook; for example, 'Sometimes you forget that LabVIEW chooses which section of code to execute next. This may be different than what you needed, and may cause unexpected things to happen.' A few respondents felt that another downside is that the dataflow paradigm results in increased memory needs.

Distinguishing whether a respondent's comment belonged to this theme was difficult. Respondents used the term dataflow in more than one way, and some respondents likely did not know what the dataflow paradigm is. Anytime respondents seemed to focus on visualness (e.g. 'It is very easy to see the dataflow in LabVIEW'), their comments were coded in the *Readability* visual theme. Also, the positive/negative count for this theme may be somewhat artificial; in interpreting comments that dataflow is different, it is not always clear whether the respondent considers this ultimately as a positive or negative aspect of LabVIEW.

6.4. Non-Visual Theme: Power

This theme holds opinions about language level and scalability. Language level refers to the fact that a programming language can hide features of the underlying machine, thus providing new, higher levels of abstraction. Scalability refers to the possibility that a programming language may provide gains for small projects, but that, as project size/complexity increases, any gains relative to other languages may disappear.

Of the 23% of the LabVIEW respondents commenting in this theme, the majority focused on language level. In particular, they commented the fact that LabVIEW automatically manages memory (for example when dynamically allocating space for arrays); the programmer never handles pointers, deallocates memory or calculates array sizes. Several respondents commented positively, stating that, 'by eliminating the tedious

and artificial tasks which have previously been necessary for programming, such as memory management…', LabVIEW 'prevents you from burning your hands' on 'canonical pointer errors'. Other respondents felt that raising the language level has costs: for example, 'Lack of pointer support makes some data structures difficult to build and use (linked lists and trees)' and 'Although the worry about memory management is removed, so is the ability to optimize its usage. Since LabVIEW is a 'safe' environment, you do not have access to some concepts (i.e. pointers) that allow you to optimize certain algorithms.' Aside from memory management, respondents also gave other indications that higher language levels cause them certain difficulties (for example, '[LabVIEW] lacks the down and dirty control often needed to do exactly what is required'). They cited problems in trying to manipulate data at the byte level and when needing to access registers. A few respondents countered that LabVIEW solves these problems by allowing the programmer to call code written in other languages.

Regardless of the positive or negative slant of the comments about language level, they all belong in the cognitive dimension closeness of mapping (since they are ultimately based on the number of low-level concepts a programmer is required to know). Interestingly, LabVIEW's memory management provides a level of programming that not all programmers appreciated.

7. Conclusion

Any technique for collecting empirical data has associated difficulties. In the case of an opinion survey, some amount of subjectivity is involved in coding respondents' open-format answers. Moreover, one cannot assume that the respondents can accurately describe their cognitive processes. Similarly, respondents may be biased in favor of the tool that they have the most experience using. Despite such difficulties, this exploratory survey was successful in collecting interesting findings.

Judging from the overall positive/negative totals and from the ratings of LabVIEW features, the 227 respondents felt that LabVIEW is a very effective tool. Of course, this sample of respondents was self-selected from a group that is likely biased in favor of LabVIEW. Nonetheless, it is interesting that this fairly large group of people find a visual language to be helpful in real-world programming contexts.

Regarding the fundamental question of whether LabVIEW's visuals are beneficial, respondents rated the value of LabVIEW's visuals significantly higher than all other LabVIEW features rated in this survey. This perception, which is consistent with Baroth and Hartsough's observational study [8], contradicted the hypothesis that respondents would rate LabVIEW's reuse higher than visual features. Ignoring the idea that the respondents were completely wrong, two interesting possibilities present themselves for further research. First, perhaps there are important benefits due to LabVIEW's visuals. Interpretation of the visual versus textual data, however, must be tempered by the comments suggesting the impact of LabVIEW's non-visual features. Thus, the second possibility takes into consideration that the respondents' opinions about LabVIEW's visualness may have been influenced by the effects of other LabVIEW features. Perhaps the visuals provide little or no advantage—or even a disadvantage—but the costs of the visuals do not outweigh the benefits provided by LabVIEW's other features.

The second hypothesis, which stated that respondents would find expressing conditional logic easier in text than in LabVIEW, was also not supported overall. The respondents as a whole rated the ease of expressing conditional logic about the same for text and LabVIEW. However, the respondents who had relatively more experience with other languages did rate text significantly higher than LabVIEW for expressing conditional logic. These ratings are consistent with Green *et al.* [9, 10].

In Green and Petre's assessment of LabVIEW, they reported that LabVIEW's visuals provide relatively good closeness of mapping, but that viscosity and secondary notation are problematic [2]. The respondents' open-format answers reveal similar opinions. They indicate that LabVIEW strives to make all dependencies visible and to achieve good closeness of mapping (especially due to its similarity to familiar circuit design notation). In having very few hidden dependencies, programmers are faced with more visible entities to layout (problems with secondary notation), to read (problems with hard mental operations) and to move around during modifications (problems with viscosity).

The open-format answers also suggest the influences of LabVIEW's non-visual features. They indicate that substantial effort has gone into designing non-visual features with the goal of raising LabVIEW's closeness of mapping: reusable code libraries, support for building front panels (GUIs) and use of the dataflow paradigm and automatic memory management are all possible advantages. One cannot yet rule out the argument that LabVIEW's reusable libraries account for a large share of LabVIEW's impact. In fact, along with the viewpoint that LabVIEW's support for building GUIs is actually a specific example of code reuse, the argument remains an area for investigation.

Acknowledgements

Our gratitude goes to the survey respondents for their time. We thank Doug Fisher, Thomas Green and Laura Novick for their helpful feedback on our work. We thank National Instruments, particularly Lisa Wells, for providing prizes as an incentive to respondents.

References

1. T. R. G. Green (1989) Cognitive dimensions of notations. In: *People and Computers V* (A. Sutcliffe & I. Macaulay, eds) Cambridge University Press, Cambridge.
2. T. R. G. Green & M. Petre (1996) Usability analysis of visual programming environments: A 'cognitive dimensions' approach. *Journal of Visual Languages and Computing* **7,** 131–174.
3. A. F. Blackwell (1996) Metacognitive theories of visual programming: what do we think we are doing? *Proceedings of the 1996 IEEE Workshop on Visual Languages* (*VL'96*), pp. 240–246.
4. K. N. Whitley (2000) Empirical research of visual programming languages: an experiment testing the comprehensibility of LabVIEW. Ph.D. dissertation, Computer Science Department, Vanderbilt University, Nashville, TN 37235, May.
5. K. N. Whitley & A. F. Blackwell (1997) Visual programming: the outlook from academia and industry. *Proceedings of the Empirical Studies of Programmers: Seventh Workshop* (*ESP7*), pp. 180–208.
6. K. N. Whitley & A. F. Blackwell (1998) Visual programming in the wild: a survey of LabVIEW programmers. Technical Report CS-98-03, Department of Computer Science, Vanderbilt University, Nashville, TN 37235, April.
7. K. N. Whitley (1997) Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing* **8,** 9–142.

8. E. Baroth & C. Hartsough (1995) Visual programming in the real world. In: *Visual Object-Oriented Programming: Concepts and Environments* (M. Burnett, A. Goldberg & T. Lewis, eds), chapter 2, Manning Publications Co., Greenwich, CT, pp. 21–42.
9. T. R. G. Green, M. Petre & R. K. E. Bellamy (1991) Comprehensibility of visual and textual programs: A test of superlativism against the 'match-mismatch' conjecture. *Proceedings of the Empirical Studies of Programmers: Fourth Workshop* (*ESP4*), pp. 121–146.
10. T. R. G. Green & M. Petre (1992) When visual programs are harder to read than textual programs. *Proceedings of the 6th European Conference on Cognitive Ergonomics* (*ECCE 6*), pp. 167–180.
11. M. Petre & A. F. Blackwell (1997) A glimpse of expert programmers' mental imagery. *Empirical Studies of Programmers: Seventh Workshop*, *1996*, pp. 109–123.
12. A. F. Blackwell. Metaphor or analogy: how should we see programming abstractions? *Proceedings of 8th Annual Workshop of the Psychology of Programming Interest Group*, pp. 105–113.
13. M. Petre & R. Winder (1988) Issues governing the suitability of programming languages for programming tasks. *People and Computers IV: Proceedings of British Computer Society HCI'88*, pp. 199–215.
14. M. Petre (1996) Programming paradigms and culture: implications of expert practice. In: *Programming Language Choice: Practice and Experience* (M. Woodman, ed.) International Thomson Computer Press, London, pp. 29–44.
15. J. Good (1996) The 'right' tool for the task: an investigation of external representation, program abstractions and task requirements. *Empirical Studies of Programmers: Sixth Workshop*, pp. 77–98.

Appendix A: Survey Questionnaire

LabVIEW Programming Survey: A 10 Minute Questionnaire

This survey has been designed to collect observations from people who have programmed in LabVIEW (i.e. who have used LabVIEW's graphical language (G) and LabVIEW's programming environment). Our goal is to learn how different aspects of LabVIEW affect people's ability to program.

- Yes, there are prizes to be won!!
- If you are curious, you can read more about the purpose of the survey.
- All responses to the survey will be confidential.
- Thank you for taking the time to participate!

Background Information

1. Please enter your e-mail address, in case you are a prize winner or in the rare case that we need to request you to clarify an answer. You may leave this blank, if you wish.

   [ ]

   Note: Your address will be used for this survey only; we will not give your email address to any third parties.
2. What kind of programming experience do you have? *Choose the one that best applies.*
   - ○ I am (or have been) a computer/programming professional.
   - ○ Although I'm not a computer professional, programming is (or has been) part of my job.
   - ○ I teach (or have taught) computer science/programming.
   - ○ Other: [ ]

3. How much experience in programming do you have? In the left column, choose one category that best describes your overall programming experience. In the right column, choose one category that best describes your LabVIEW programming experience.

Overall Programming	LabVIEW Programming	
○	○	I have only played around with programming.
○	○	I have taken a training course to learn programming.
○	○	I have written several small programs.
○	○	I have written medium-sized programs.
○	○	I have worked on big projects over a year or more.
○	○	I have worked as a programmer for several years.
○	○	I have spent more than 10 years programming fulltime.
○	○	Other:

4. What programming language(s) do you have the most experience using? *Note that this question includes, but is not limited to, your experience with LabVIEW.*

LabVIEW

5. What is your overall opinion of LabVIEW?

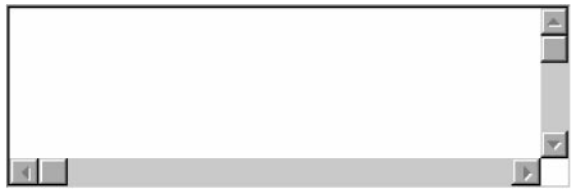
6. Please rate the following LabVIEW features on a scale from 1 to 6 according to how much each is an advantage or disadvantage of using LabVIEW. Interpret the scale as follows:

Big Disadvantage	Disadvantage	Slight Disadvantage	Slight Advantage	Advantage	Big Advantage	Not Applicable
1	2	3	4	5	6	N/A

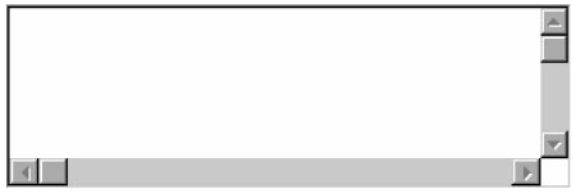
The 'Not Applicable' column should be used only if a LabVIEW feature is not relevant to the type of LabVIEW programming that you do.

	1	2	3	4	5	6	N/A
LabVIEW Toolkits/Reusable VIs	○	○	○	○	○	○	○
National Instruments Hardware	○	○	○	○	○	○	○
LabVIEW's Graphical Language (G)	○	○	○	○	○	○	○
G's Use of Dataflow Programming	○	○	○	○	○	○	○
LabVIEW's Support for "Front Panel" Interfaces	○	○	○	○	○	○	○
Customer Service for LabVIEW	○	○	○	○	○	○	○
Other:	○	○	○	○	○	○	
Other:	○	○	○	○	○	○	

7. Comparing LabVIEW to other programming languages, can you provide examples of how LabVIEW makes certain programming tasks easier to accomplish?



8. Comparing LabVIEW to other programming languages, can you provide examples of how LabVIEW makes certain programming tasks more difficult to accomplish?



LabVIEW's Graphical Language (G)

9. We would like you to compare LabVIEW's graphical language, G, to a textual programming language that you frequently use. This question is divided into two charts: you will first rate the textual language of your choice; then, you will rate G along the same dimensions. *If you do not have experience using a textual programming language, please complete the rating chart for the textual language according to how you would expect a textual language to be.*

What textual language are you comparing to G? *If you are not familiar with a textual language*, *write* '*Guess*'.

My **textual** language is:

	1	2	3	4	5	6	
weak	○	○	○	○	○	○	powerful
hard to write	○	○	○	○	○	○	easy to write
hard to comprehend (i.e., unreadable)	○	○	○	○	○	○	easy to comprehend (i.e., readable)
hard to modify	○	○	○	○	○	○	easy to modify
irritating	○	○	○	○	○	○	enjoyable
does not handle repetitive logic (loops) easily	○	○	○	○	○	○	handles repetitive logic (loops) easily
does not handle conditional logic easily	○	○	○	○	○	○	handles conditional logic easily
Other:	○	○	○	○	○	○	
Other:	○	○	○	○	○	○	

LabVIEW's graphical language (G) is:

	1	2	3	4	5	6	
weak	○	○	○	○	○	○	powerful
hard to write	○	○	○	○	○	○	easy to write
hard to comprehend (i.e., unreadable)	○	○	○	○	○	○	easy to comprehend (i.e., readable)
hard to modify	○	○	○	○	○	○	easy to modify
irritating	○	○	○	○	○	○	enjoyable
does not handle repetitive logic (loops) easily	○	○	○	○	○	○	handles repetitive logic (loops) easily
does not handle conditional logic easily	○	○	○	○	○	○	handles conditional logic easily
Other:	○	○	○	○	○	○	
Other:	○	○	○	○	○	○	

10. How and why does the graphical nature of G affect the "brain-work" involved in programming?

Other Comments

11. Do you have any comments about the survey itself or about how LabVIEW affects the programming process? For example, please tell us if you think that any of the survey questions are not relevant to assessing LabVIEW's performance.

Before submitting your survey responses, please consider whether you have any additional information that you could provide in Questions 7 and 8 (about the advantages and disadvantages of LabVIEW) or in Question 10 (about how G affects the "brain-work" of programming). Your answers to these questions would help us greatly in our analysis of this survey.