Handbook of Software Engineering and Knowledge Engineering
Vol. 2 (2001)
© World Scientific Publishing Company

SOFTWARE ENGINEERING FOR VISUAL PROGRAMMING LANGUAGES

MARGARET BURNETT
Department of Computer Science
Oregon State University
Corvallis, OR 97331
E-mail: burnett@cs.orst.edu

This article describes emerging research into how visual programming languages may lead to and even require the development of new software engineering support paradigms.

Keywords: visual programming languages, software engineering, end-user programming, end-user software engineering, Forms/3.

1. Introduction

Visual programming languages (VPLs) are becoming increasingly common in several domains. For example, visual programming languages or sublanguages are becoming the most common way to do some kinds of GUI programming, are becoming the most common way of specifying visualization graphics depicting scientific data, and are also starting to appear as macro generators for end-user applications. However, despite the increase in the use of VPLs for these and other programming tasks, there has been almost no attention to software engineering support mechanisms when working in these languages.

Visual programming is programming in which more than one dimension is used to convey semantics [3]. Examples of such additional dimensions are the use of multidimensional objects, the use of spatial relationships, or the use of the time dimension to specify “before-after” semantic relationships. Each such potentially significant object or relationship is a token (just as in traditional textual programming languages each word is a token) and the collection of one or more such tokens is a *visual expression*. Examples of visual expressions used in visual programming include diagrams, free-hand sketches, icons, or demonstrations of actions performed by graphical objects. When a programming language’s (semantically significant) syntax includes visual expressions, the programming language is a *visual programming language*.

VPLs often include text in a multidimensional way. Traditional textual programming languages sometimes also incorporate a little two-dimensional syntax for text, but only in very limited forms.[1] Thus, multidimensionality is the essential difference between VPLs and strictly textual languages.

This point—that the difference between programming languages that are VPLs and those that are not comes down simply to multidimensionality—seems to predict that software engineering methodologies and devices developed for software written in non-visual programming languages should serve for software written in VPLs as well. However, multidimensionality has led VPLs researchers into new programming frontiers, and there have so far been only the beginnings of accompanying software engineering research into these frontiers. This article surveys several such beginnings using as an example VPL the research visual spreadsheet language Forms/3 [5].

We begin with a discussion of software engineering issues particular to VPLs, and then survey how these issues affect the following subareas of software engineering for VPLs: supporting program comprehension in visual programs, testing visual programs, debugging visual programs, and reusing visual code.

[1] For example, in traditional languages the x-dimension connects a linear string in the language, but the y-dimension may allow optional line spacing as a documentation device or for limited semantics (such as “continued from previous line”). Here only one of these dimensions truly conveys semantics, and the second dimension has been limited to a teletype notion of spatial relationships so as to be expressible in a one-dimensional string grammar.

2. Software engineering issues particular to VPLs

Three features alluded to above have particular implications to software engineering in VPLs. The first is diversity of audience: while some users of VPLs are professional programmers, some are end users with no training in professional software engineering notions and methods.

The second is how to develop software engineering approaches that are fully compatible with the non-traditional paradigms and mechanisms used in VPLs. For example, specifying program semantics by directly manipulating objects or demonstrating with concrete examples is not the way programming is done in traditional languages, and there is as yet little research into suitable software engineering support mechanisms for these non-traditional language features.

The third is more an opportunity than a problem: many of the language features made possible by multidimensionality and pioneered in VPLs have inherent elements of software engineering support. The most common of these features are *concreteness*, *directness*, *explicitness*, and *immediate visual feedback*.

Concreteness means expressing some aspect of a program using particular instances, such as specifying some aspect of semantics by specifying desired behavior using a specific object or value. A software engineering implication of concreteness is that, because testing is also done in terms of concrete values, concreteness during program development automatically performs incremental testing on at least one test value.

Directness in the context of direct manipulation is usually described as “the feeling that one is directly manipulating the object” [18]. From a cognitive perspective, directness in computing means a small distance between a goal and the actions required to achieve the goal [7, 8, 10]. In VPLs, both of these definitions are relevant. As Green and Petre point out in their distillation of psychology-of-programming literature for VPL designers, since “programming requires mapping between a problem world and a program world, the closer the programming world is to the problem world, the easier the problem solving out to be” [7]. An example of directness in a VPL would be manipulating an object to specify its change in location—that is, using movement to specify movement, rather than using functions on numbers (x and y screen coordinates) to specify it.

Some aspect of semantics is *explicit* in the environment if it is directly stated (textually or visually), without the requirement that the programmer infer it. An example of explicitness in a VPL would be explicit depiction of dataflow relationships via edges among related program objects. Such drawings are common software engineering support devices for traditional languages, but only as separate tools that must be invoked in some step separated from the editing or debugging process. In VPLs, such drawings can be part of the language syntax or an automatic side effect maintained by the environment.

In the context of visual programming, *immediate visual feedback* refers to automatic display of effects of program edits. Tanimoto has coined the term *liveness*, which categorizes the immediacy of semantic feedback that is automatically provided during the process of editing a program [21]. An example of a high degree of liveness is the automatic recalculation feature of spreadsheets. This feature provides some of the functionality of traditional debuggers, but is much more incremental.

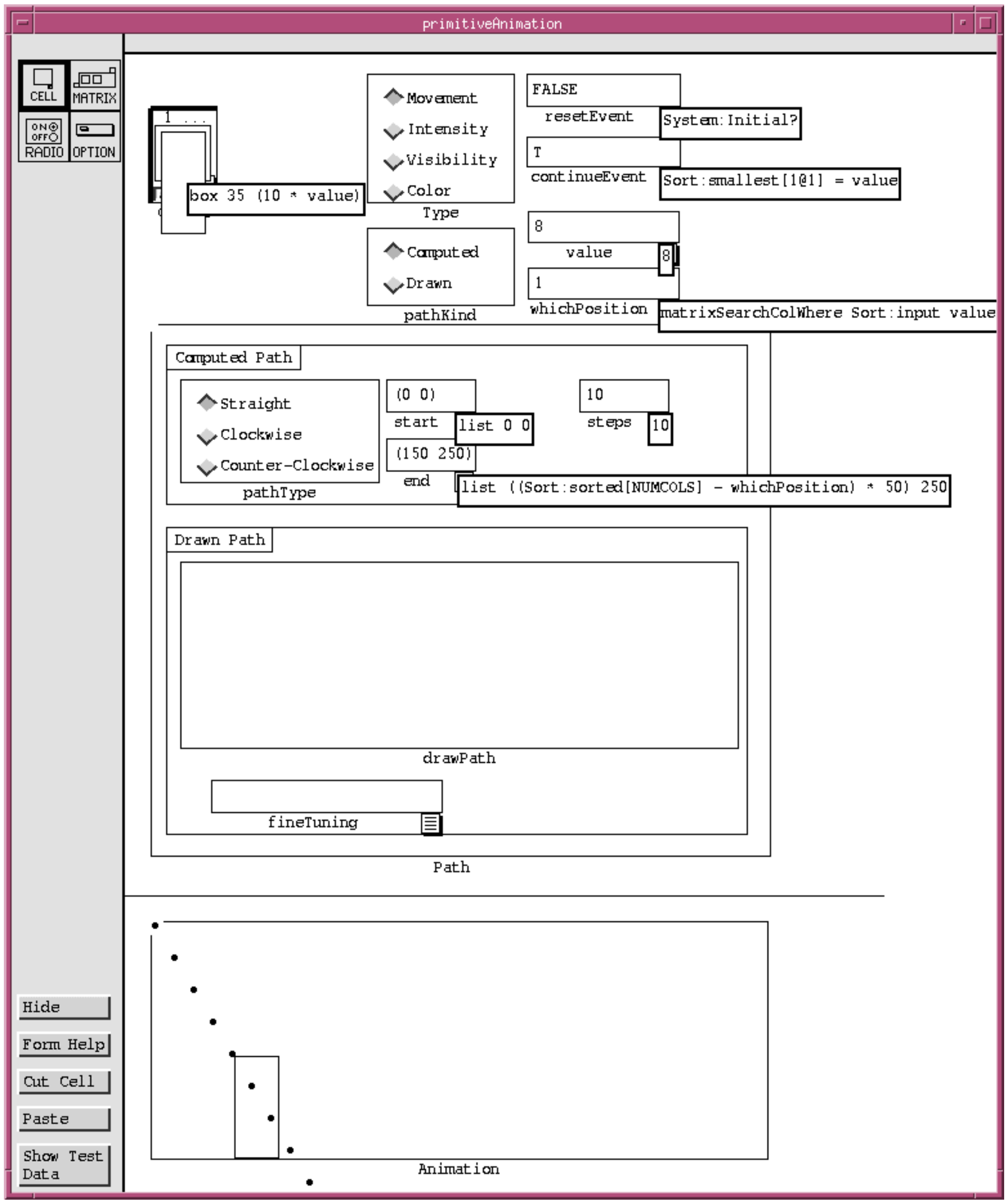
Given VPLs’ unique features, the challenge is to develop software engineering approaches that are compatible with these features and take advantage of the opportunities they offer, yet have the power and rigor of traditional approaches.

3. Dynamic documentation and program comprehension

Program comprehension is critical in the debugging and maintenance phases of the software lifecycle. The combination of concrete values with visual feedback leads naturally to the idea of software animation, and many VPLs include various forms of this. Software animation can be thought of as a dynamically computed documentation mechanism for supporting program comprehension. However, a difference between software animation and separately prepared documentation is that, since the documentation-oriented code is tied to the logic-oriented code, software animations do not become out of date as the source code evolves.

Several empirical studies have been done on VPLs’ effects on program comprehension. See Whitley [24] for a survey of this work. The results of these studies have been mixed, reporting findings for some kinds of programs or audiences in which VPLs and/or visual notations are linked with greater comprehension, and others in which strictly textual languages and/or notations have been linked with greater comprehension. Unfortunately, almost all of these studies have investigated only static diagrammatic notations, omitting the dynamic graphics, concreteness, and feedback mechanisms found in many VPLs. However, the use of dynamic graphics of concrete values as a program comprehension mechanism was specifically investigated in two studies regarding software animation's effects on people's ability to comprehend and work with previously existing programs [9, 20]. The results were that animations helped the participants with program comprehension if the participants were actively involved in working with the animations.

The type of animation investigated in these two studies was graphical animation of textual programs, but, as pointed out above, VPLs often include similar capabilities simply as by-products. In Forms/3, animation follows naturally from the combination of support for graphical types, evolving data values, and incremental visual feedback as computations evolve and program edits are made [5]. For example, to provide animated documentation of a selection sort, a programmer may wish to emphasize the "move" portion of the algorithm, having each element glide down the screen to its new location. To specify such an animation, the programmer gives formulas for the intermediate positions through which a graphical depiction of an element should travel, either by specifying straight/clockwise/counter-clockwise and the start, end, and number of steps, or by directly drawing the path (middle of Figure 1). When the user provides formulas in this form to create an animation of one element of a matrix being sorted, the system automatically generalizes it for the remaining elements of the matrix [6]. After this generalization of Figure 1, the result is as shown in Figure 2. For animation effects other than spatial movement, the programmer can select options on the animation form to specify paths through "visibility space" (for fade-in/fade-out sequences), through "color space" (for gradual color transitions), or through "intensity space" (for brightening/dimming transitions).



box 35 (10 * value)

Type: [✓ Movement] Intensity Visibility Color

pathKind: [✓ Computed] Drawn

resetEvent: FALSE — System:Initial?

continueEvent: T — Sort:smallest[1@1] = value

value: 8 — 8

whichPosition: 1 — matrixSearchColWhere Sort:input value

pathType: [✓ Straight] Clockwise Counter-Clockwise

start: (0 0) — list 0 0

end: (150 250) — list ((Sort:sorted[NUMCOLS] - whichPosition) * 50) 250

steps: 10 — 10

Fig. 1. An Animation form for one element of the selection sort animation. The parameters are established in cell formulas at the top and middle (through a flexible combination of text and/or drawing), and the result is at the bottom. Automatic generalization of the formula that references this form causes, on a lazy basis, a copy of this form to be created to animate whichever element is actively being sorted.

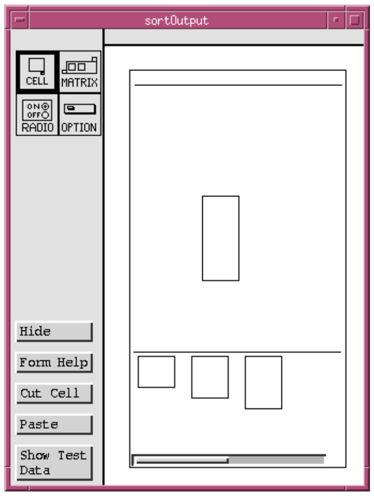


Fig. 2. A sort animation shows the elements of the unsorted group at the top being moved one at a time to the sorted group at the bottom. The final element is moving from the top left corner to the bottom right at this point in the animation.

4. Testing

The high degrees of concreteness and immediate visual feedback present in some VPLs have been motivated in part by a kind of "instant testing" goal, with the idea that if the user immediately sees the result of a program edit, he or she will spot programming bugs as quickly as they are made, and hence will be able to eradicate them right away. This motivation has been especially prevalent in end-user VPLs, but is also found in VPLs for professional programmers.

However, spreadsheet systems provide evidence that concreteness and immediate visual feedback have not been enough for finding and removing most of the bugs in a spreadsheet's formulas: there is a substantial body of research showing that spreadsheets often contain bugs. For example, field audits of real-world spreadsheets have found that 20-40% of these contain bugs, and that between 1% and 4% of all cells contain bugs [22]. Also, in an early empirical study of experienced spreadsheet users, 44% of the spreadsheets created by those users were found to contain user-generated bugs [2]. Results of several later studies have been similar: between 10% and 90% of the spreadsheets examined have been found to contain bugs. (See Panko and Halverson [11] for a survey of these studies). Compounding this problem, creators of programs in language environments featuring immediate visual feedback, such as in spreadsheet systems, seem to express unwarranted confidence in the reliability of their programs [25].

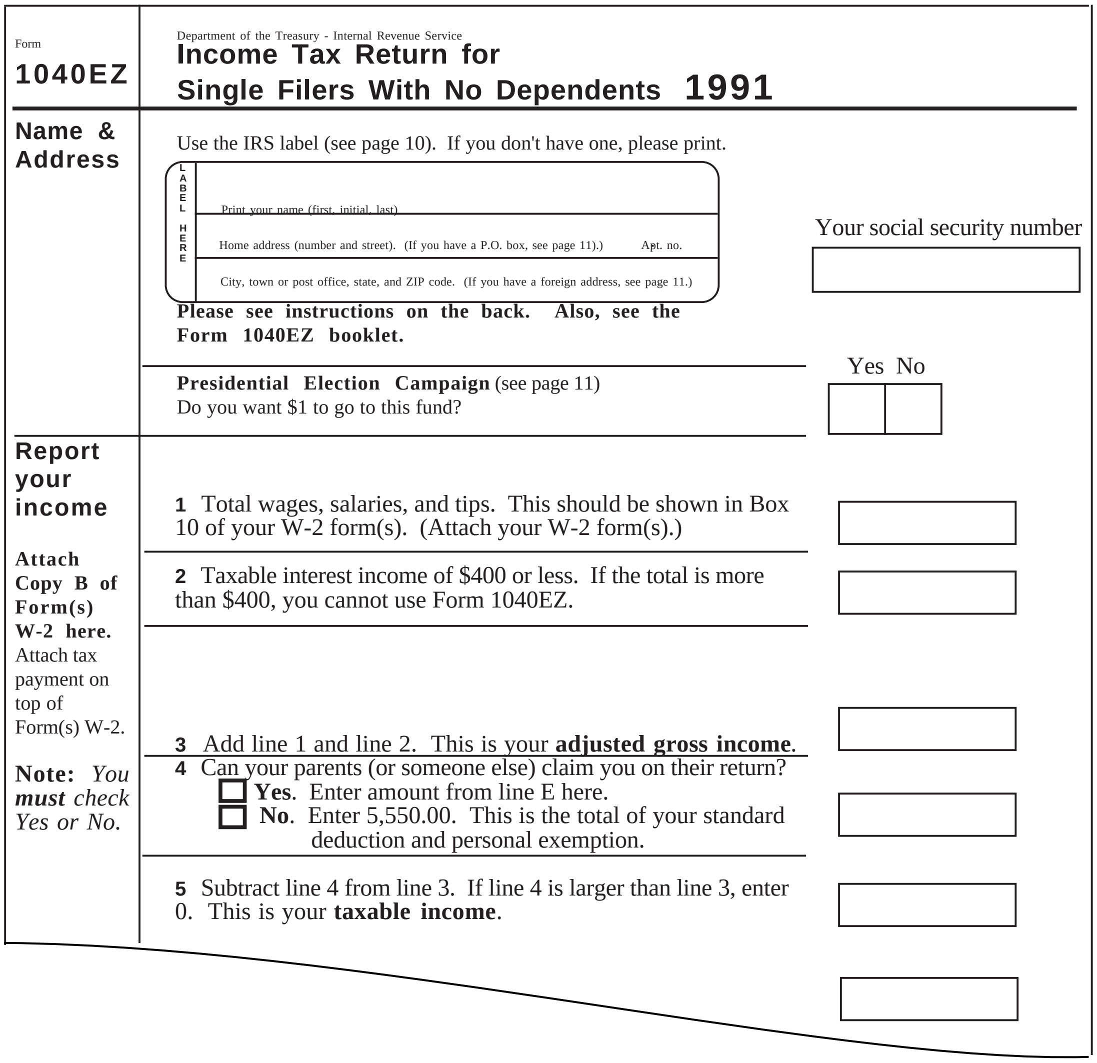
There is recent work in Forms/3 to bring some of the benefits of applying formalized notions of testing to the informal, incremental, development world of spreadsheet-like VPLs, including a range of research languages in this paradigm as well as standard commercial spreadsheet packages [4, 15, 16]. The "What You See Is What You Test" (WYSIWYT) methodology is an approach to testing for highly visual problem-solving environments such as spreadsheets. The methodology is completely visual, and is designed to support end users as well as more sophisticated programmers.

In software engineering research, a definition of what it means for a program to be tested enough is called a *test adequacy criterion*. Under the WYSIWYT approach, the VPL's designer chooses a test adequacy criterion, such as that each pairing of subexpression relationships must be exercised by at least one test. Using the test adequacy criterion to define the ideal, the system continuously communicates to users how closely they have gotten, through their testing activities, to this ideal. To do this, the system treats each user "decision" about correctness as a test, and the user communicates those decisions to the system by checking off a value whenever he or she notices that it is correct. The system tracks these tests and their implications, and also keeps track of what previous tests need to be redone as a result of formula edits. This approach provides feedback about testing adequacy at all stages of spreadsheet development, with the intent of helping users detect bugs in their spreadsheets.

The methodology has been prototyped in Forms/3, but to best illustrate how an end user might use it, the following scenario illustrates instead how the WYSIWYT methodology might appear if integrated into widely used commercial spreadsheet packages.

Suppose an end user has a printout of a tax form such as in Figure 3 in front of her, and she wants to use a spreadsheet to figure out the answers. To do this, she has created a spreadsheet such as the one in Figure 4.

Although this spreadsheet is simple, there are several ways the user could end up reporting the wrong answer. Like many taxpayers, she may be struggling to gather up all the required data, and may change her mind about the right data values to enter. If she has been taking shortcuts with the formulas, basing them upon the conditions present in her first version of the data (such as not bothering to use a *max* operator in line 5 to prevent negatives), the formulas are probably not very general, and may cause problems if her data changes. For example, if she entered "line 4 - line 3" as the formula for line 5, but later changes line 4 to 5500 because her parents tell her they did not claim her this year after all, then the formula for line 5 will not give the correct answer. Similar problems could arise if she discovers that she entered data from the wrong box of her W-2, and so on.



Form **1040EZ**

Department of the Treasury - Internal Revenue Service
Income Tax Return for Single Filers With No Dependents 1991

Name & Address

Use the IRS label (see page 10). If you don't have one, please print.

LABEL HERE

Print your name (first, initial, last)

Home address (number and street). (If you have a P.O. box, see page 11).) Apt. no.

City, town or post office, state, and ZIP code. (If you have a foreign address, see page 11.)

Your social security number

Please see instructions on the back. Also, see the Form 1040EZ booklet.

Presidential Election Campaign (see page 11)
Do you want $1 to go to this fund?

Yes No

Report your income

Attach Copy B of Form(s) W-2 here. Attach tax payment on top of Form(s) W-2.

Note: *You* ***must*** *check Yes or No.*

1 Total wages, salaries, and tips. This should be shown in Box 10 of your W-2 form(s). (Attach your W-2 form(s).)

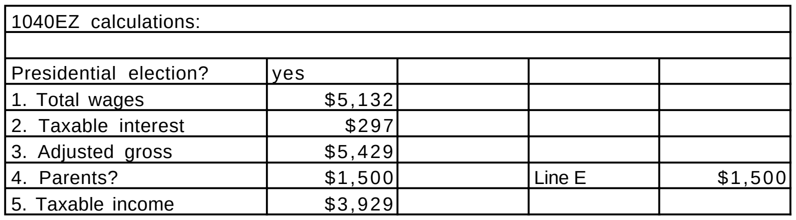
2 Taxable interest income of $400 or less. If the total is more than $400, you cannot use Form 1040EZ.

3 Add line 1 and line 2. This is your **adjusted gross income**.

4 Can your parents (or someone else) claim you on their return?
☐ **Yes**. Enter amount from line E here.
☐ **No**. Enter 5,550.00. This is the total of your standard deduction and personal exemption.

5 Subtract line 4 from line 3. If line 4 is larger than line 3, enter 0. This is your **taxable income**.

Fig. 3. A portion of a tax form.



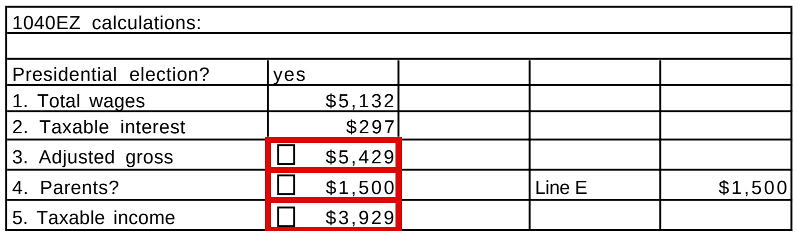
1040EZ calculations:				
Presidential election?	yes			
1. Total wages	$5,132			
2. Taxable interest	$297			
3. Adjusted gross	$5,429			
4. Parents?	$1,500		Line E	$1,500
5. Taxable income	$3,929			

Fig. 4. The user's spreadsheet to figure out the taxes. The first few cells are simply data values. Line 3's formula is line 1 + line 2, line 4's formula is a reference to line E, and line 5's formula is line 3 - line 4.

Even in this simple case, the WYSIWYT methodology can provide benefits. Figure 5 shows a mock-up of how it might be incorporated into a popular spreadsheet package. All cells containing formulas (as opposed to data values) are initially red-bordered with checkboxes, as in Figure 5(a). The first time the user sees a red border, she moves her mouse over it and the tool tips inform her that "red borders mean untested and blue borders mean tested. You can check cells off when you approve of their values." The user checks off a value that she is sure is correct, and a checkmark (√) appears in the cell the user checked off, as in Figure 5(b). Checkmarks are used to show where the user has *explicitly* checked off a value. The wider *implications* of this checkmark are reflected by border colors. Thus, the border of this explicitly approved cell and of cells contributing to it become blue. If she then changed some data, any affected checkmarks would be replaced with question marks (?). This would remind her to check again the cells whose values she thought were important enough to check off before.

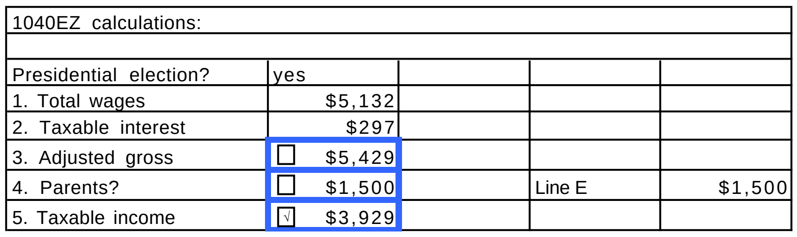
Now suppose that, instead of replacing a data value, the user makes the formula change in line 4 alluded to above, changing the previous formula to the constant 5500 instead of the former reference to line E. Since the change she made involved a formula (the one she just changed to a data value), the affected cells' borders revert to red and downstream √s and ?s disappear, indicating that these cells are now completely untested again. See Figure 5(c). The maintenance of the "testedness" status of each cell throughout the editing process, as illustrated in Figure 5(c), is an important benefit of the approach. Without this feature, the user may not realize that the testing she did before became irrelevant with her formula change and now needs to be redone.

A primary goal of this approach is to reduce overconfidence about the correctness of spreadsheet formulas. In an empirical study, the methodology significantly reduced overconfidence about how tested spreadsheets were, as well as improving effectiveness and efficiency of testing [17].



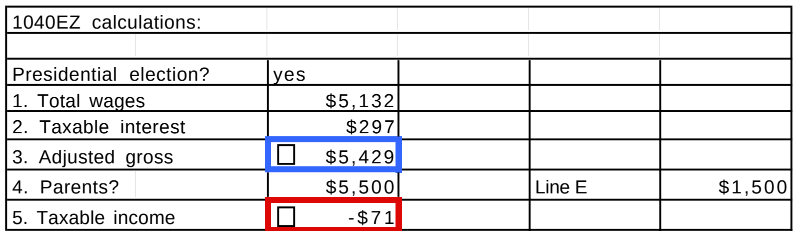
1040EZ calculations:				
Presidential election?	yes			
1. Total wages	$5,132			
2. Taxable interest	$297			
3. Adjusted gross	☐ $5,429			
4. Parents?	☐ $1,500		Line E	$1,500
5. Taxable income	☐ $3,929			

(a)



1040EZ calculations:				
Presidential election?	yes			
1. Total wages	$5,132			
2. Taxable interest	$297			
3. Adjusted gross	☐ $5,429			
4. Parents?	☐ $1,500		Line E	$1,500
5. Taxable income	☑ $3,929			

(b)



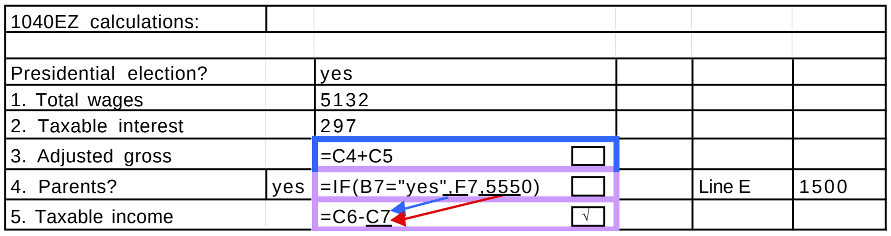
1040EZ calculations:				
Presidential election?	yes			
1. Total wages	$5,132			
2. Taxable interest	$297			
3. Adjusted gross	☐ $5,429			
4. Parents?	$5,500		Line E	$1,500
5. Taxable income	☐ -$71			

(c)

Fig. 5. A mock-up of a popular spreadsheet package if enhanced by the WYSIWYT technology. In this black-and-white shot, red shows a dark gray and blue shows as lighter gray.
(a): All cells containing formulas are initially red, meaning untested.
(b): Whenever the user makes a decision that some data value is correct, she checks it off. The checkmark appears in the cell she explicitly validated. Further, all the borders of cells contributing to that correct value become more tested (closer to pure blue). This example has such simple formulas, all the cells at this point are pure blue, meaning fully tested.
(c): The user changes the formula in line 4 to a constant. This change causes affected cell in the bottom row to be considered untested again; hence it is now pure red.

The next year, the user may want to improve the spreadsheet so that she can use it year after year without having to redesign each formula in the context of the current year's data values. For example, she adds the yes/no box from the tax form's line 4 to her spreadsheet's line 4 and uses the *if* operator in the formula for line 4. Because of this *if*, she will need to try at least two test cases for line 4's cell to be considered tested: one that exercises the "*yes*" case and one that exercises the "*no*" case. (See Rothermel et al. [15, 16] for descriptions of the coverage criteria currently in use as well as other possible criteria that can alternatively be employed.)

Because of this, when the user checks off one data value as in Figure 6, the border for lines 4 and 5 turn purple (50% blue and 50% red). To figure out how to make the purple cells turn blue, the user selects one of them and hits a "show details" button. The system then draws arrows pertaining to the subexpression relationships, with colors depicting which cases still need to be tested. The arrow from the last subexpression is red, pointing out that the "no" case still needs to be tried.



1040EZ calculations:					
Presidential election?		yes			
1. Total wages		5132			
2. Taxable interest		297			
3. Adjusted gross		=C4+C5 ☐			
4. Parents?	yes	=IF(B7="yes",F7,5550) ☐		Line E	1500
5. Taxable income		=C6-C7 √			

Fig. 6. Some cells require more than one test value to become completely tested, as this formula view with purple cell borders and red and blue arrows between subexpressions shows. The adjusted gross value is blue-bordered, the two below it are purple, the arrow from F7 is blue, and the arrow from 5550 is red.

5. Debugging

We pointed out earlier that the concreteness and immediate visual feedback features present in a number of VPLs provide some of the functionality needed for debugging. However, the prevalence of formula errors in spreadsheets shows that this functionality, valuable though it is, is not enough to either keep the errors out in the first place, or to get the errors out once they get in.

In order to begin debugging, a user must first know that something is wrong. The animations (dynamic documentation) described in Section 3 are one potential way VPLs can use to help the user spot incorrect behavior. Another possibility is that, during the testing interactions described in Section 4, the user comes across a value that is not correct.

After the user has detected the presence of an error, debugging is classically said to have three stages: locating the error, fixing it, and then verifying that the fixed portion is now correct. To provide support for locating the error, the WYSIWYT methodology has been extended as follows. In addition to marking values correct with a checkmark, a user can mark a value incorrect by X'ing it out. The dataflow path contributing to the X'd out value is the portion of the spreadsheet in which the fault exists, and this is highlighted to the user; see Figure 7. The technique used to decide which cells to highlight is a set of heuristics based on ideas from research in slicing and dicing [12].

Forms/3 does not explicitly support the fixing stage, other than to follow the VPL common practice of allowing formulas to be edited incrementally without requiring separate tools for editing, compiling, etc. However, the verify stage is the same idea as regression testing, which is supported by the WYSIWYT methodology. In Forms/3, if a cell is edited, downstream cell colors, border colors, checkmarks, and question marks are reset, making explicit to the user which cells need to be re-examined at this point.

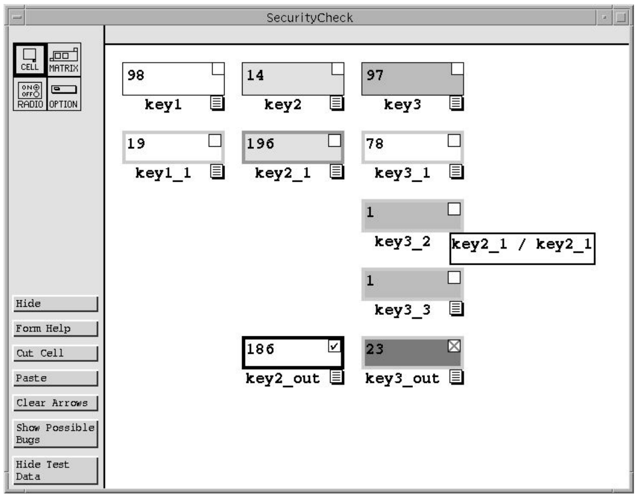


Fig. 7. An incorrect value is present in cell key3_out, indicated by an X in the checkbox, placed by the user. The system can display the sub-slice in which the fault lies (highlighted cells). Each cell's likelihood of containing the fault is communicated by the darkness of its background.

6. Code reuse

Although the idea of reusing code is very appealing, effective reuse has long been acknowledged as a problem. To help address the difficulties, many advocate a strong management commitment to code reuse, leading to the treatment of code as an asset to be carefully managed in a well organized repository. However, the advent of the web may bring a change to this outlook, encouraging informal, loosely organized code repositories. This may be particularly true for the VPLs that are aimed at end users, whose software creations are not usually managed by anyone other than themselves. In fact, some recent commercial end-user VPLs such as AgentSheets [13] and Stagecast (previously known by the names Cocoa and KidSim) [19] already provide just such repositories.

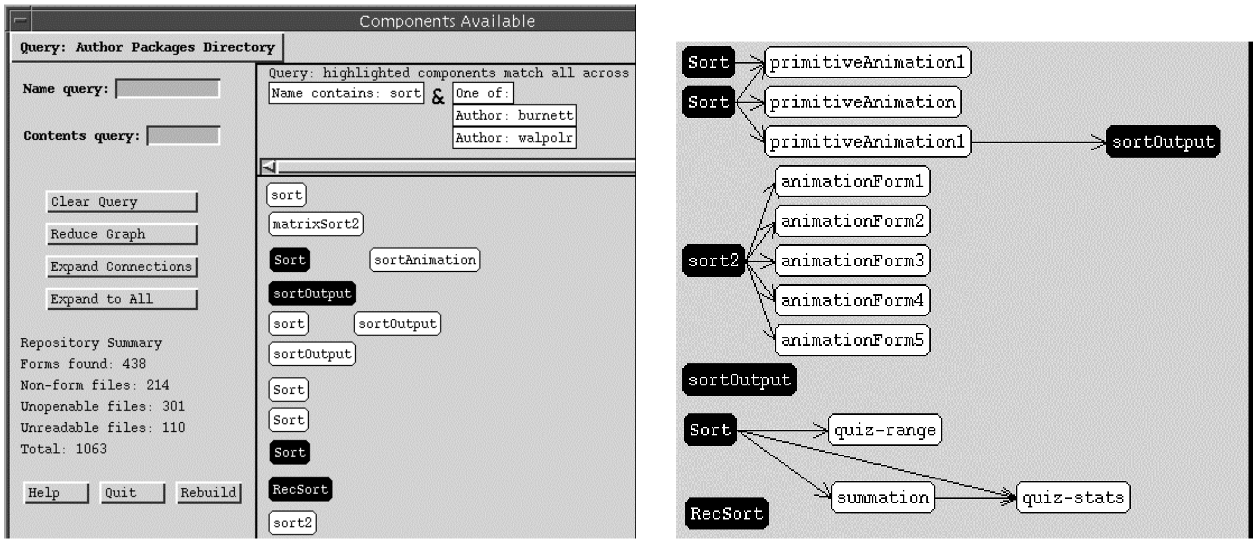
This discussion of reuse will be confined to the use of an existing code component in place of creating a new component. In general, a *component* is any artifact of the software process, but this article concentrates solely on code components. The term *repository* will be used to mean a collection of components. To distinguish between reuse tasks, even if they are performed by the same person, the term *producer* will mean the programmer who is building reusable components, and the term *consumer* will mean the programmer who is interested in using these components. Finally, *packaging* tasks are the work traditionally required of producers to prepare a component for inclusion in a formally controlled code repository, such as conforming to standards, preparing documentation, providing test suites, etc.

In this kind of environment, it is not realistic to expect the producers of the software (end users) to follow any sort of rigorous, packaging-for-reuse standards in making their code available for use by others. Thus, if reuse is to be supported, more sophisticated support mechanisms must be made available on the consumer side. In other words, the following, highly useful assumptions that have formed the basis of traditional approaches to supporting code reuse cannot be assumed here because informal repositories simply do not have them:

- A managing body to enforce standards and oversee the repository.
- A set repository structure such as a hierarchy of component categories or collections.
- Component producers who meticulously provide component packaging to aid the consumer.

Biggerstaff and Richter [1] identify four fundamental reuse problems a successful reusability system must address: finding components, understanding components, modifying components, and composing components. But given a constantly evolving repository with no requirement for classification or packaging by the code producer, how can a VPL support these needs?

Finding components has been the area best supported by traditional software engineering support mechanisms, namely query facilities allowing consumers to search by keyword, author, comment contents, etc. Some VPLs have employed these mechanisms as well, and in fact have easily taken them a step forward: by depicting their availability explicitly in the programming environment, a VPL can provide a continual reminder that reuse opportunities exist. For example, the environment of Forms/3 automatically provides a display and query facility for code components available for incorporation and/or specialization into an evolving program [23], as shown in Figure 8.


```
Query: Author Packages Directory
Name query:
Contents query:
Query: highlighted components match all across
Name contains: sort  &  One of:
                        Author: burnett
                        Author: walpolr

Repository Summary
Forms found: 438
Non-form files: 214
Unopenable files: 301
Unreadable files: 110
Total: 1063
```

Fig. 8. (Left): Forms/3 has a query facility to help users find potentially useful code components. (Right): Interrelationships with other components are shown. Arrows leading out of a node give examples of how other components have used this component.

Having found a possibly reusable component, a consumer's next step will be to understand enough about its behavior to determine whether to consider making use of it. This is an area of opportunity for VPLs, because if a VPL includes aids to program comprehension, and if the VPL also features tightly integrated encouragement for reuse, then the two can work together to form a "whole bigger than the sum of its parts." For example, in Forms/3, if the consumer is interested in a sorting component, he or she can search for it using the query facility in Figure 8 and, upon clicking on the animation nodes, can watch the dynamic documentation of the sort described in Section 3.

Modifying and composing components are the ways consumers actually make use of components they have selected. A recent study of professional Smalltalk users [14] observed that consumers make extensive use of previous usage contexts when figuring out how to use unfamiliar components. The right side of Figure 8 shows how the two-dimensional repository display with explicit depiction of relationships with other components helps to facilitate finding examples of how to use it. As with the left side of the figure, touching the quiz-stats node brings into view the executable source code of quiz-stats and its supporting nodes, where the user can watch its execution, provide different sample values and try again, and modify or copy portions of quiz-stats as needed for their own purposes. Once a user begins making such changes to incorporate the new component, the testing and debugging features discussed earlier can support this effort.

This ability to browse, retrieve and experiment with usage contexts seems critical to allowing consumers to successfully reuse even code that has not been specially packaged by its producers. This ability permits an alternative to the documentation and standardization procedures traditionally required of producers, which is important in informal repositories and in cooperative repositories not "owned" by any organization.

7. Conclusion

Three features of VPLs raise challenges and opportunities regarding how to provide appropriate software engineering support. The first challenge comes from the fact that non-traditional audiences are often the target users of these languages, and these users are usually not formally trained in programming or software engineering. The second arises because many VPLs include unusual features that do not seem to directly map to previous software engineering support mechanisms. Such features include use of concrete objects, demonstrations, non-imperative paradigms, and multidimensionality to specify program semantics. At the same time, these features also give rise to opportunities for highly integrated software engineering support mechanisms. By attending to the significant differences between traditional languages and VPLs, it may be possible to devise software engineering support methods that achieve the desirable properties of methods for traditional languages, although they may look very different from traditional methods.

Acknowledgments

This work was supported in part by the National Science Foundation under CCR-9806821 and NYI Award CCR-9457473 and by Hewlett-Packard. The material on software animation in Forms/3 summarizes collaborative work by the author with Paul Carlson and J. J. Cadiz. The material on code reuse in Forms/3 summarizes collaborative work by the author with Rebecca Walpole Djang, Martin Griss, and Gregson Siu. The material on the WYSIWYT methodology summarizes collaborative work by the author with Gregg Rothermel, Lixin Li, James Reichwein, Andrei Sheretov, Curtis Cook, Thomas Green, and Karen Rothermel, with assistance and contributions from several graduate students in the Oregon State University Visual Programming Research Group; patents pending.

References

1. T. Biggerstaff and C. Richter. Reusability Framework, Assessment, and Directions. In T. J. Biggerstaff and A. J. Perlis, (eds.), *Software Reusability: Concepts and Models, Volume 1* (Addison-Wesley, Reading, MA, 1989).
2. P. Brown and J. Gould, Experimental study of people creating spreadsheets, *ACM Transactions on Office Information Systems* **Vol.5** (1987) 258-272.
3. M. Burnett, Visual Programming, In J. Webster, (ed.), *Encyclopedia of Electrical and Electronics Engineering* (John Wiley & Sons, 1999).
4. M. Burnett, A. Sheretov, and G. Rothermel, Scaling Up a 'What You See Is What You Test' Methodology to Testing Spreadsheet Grids, *1999 IEEE Symposium on Visual Languages* (Sept. 13-16, 1999) 30-37.
5. M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Reichwein, and S. Yang, Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm, *Journal of Functional Programming* (to appear).
6. P. Carlson, M. Burnett, and J. Cadiz, A Seamless Integration of Algorithm Animation into a Visual Programming Language, *ACM Proceedings of AVI'96, International Workshop on Advanced Visual Interfaces* (May 1996) 194-202.
7. T. Green and M. Petre, Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing* **Vol.7, No.2** (June 1996) 131-174.
8. E. Hutchins, J. Hollan, and D. Norman, Direct Manipulation Interfaces. In D. Norman and S. Draper (eds.), *User Centered System Design: New Perspectives on Human-Computer Interaction* (Lawrence Erlbaum Assoc., Hillsdale, NJ, 1986) 87-124.
9. A. Lawrence, A. Badre, and J. Stasko, Empirically Evaluating the Use of Animations to Teach Algorithms, *1994 IEEE Symposium on Visual Languages*, (Oct. 4-7, 1994) 48-54.
10. B. Nardi, *A Small Matter of Programming: Perspectives on End User Computing* (MIT Press, Cambridge, MA, 1993).
11. R. Panko and R. Halverson, Spreadsheets on Trial: A Survey of Research on Spreadsheet Risks, *Hawaii International Conference on System Sciences* (Jan. 2-5, 1996).
12. J. Reichwein, G. Rothermel, and M. Burnett, Slicing Spreadsheets: An Integrated Methodology for Spreadsheet Testing and Debugging, *Conference on Domain Specific Languages (DSL'99)*, (Oct. 3-5, 1999) 25-38.
13. A. Repenning and J. Ambach. Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition and Sharing, *IEEE Symposium on Visual Languages* (Sept. 1996) 102-109.
14. M. Rosson and J. Carroll, The Reuse of Uses in Smalltalk Programming, *ACM Trans. on Computer-Human Interaction* **Vol.3, No.3**, (Sept. 1996) 219-253.
15. G. Rothermel, L. Li, C. DuPuis, and M. Burnett, What you see is what you test: A Methodology for Testing Form-Based Visual Programs, *International Conference on Software Engineering* (April 1998) 198-207.
16. G. Rothermel, M. Burnett, L. Li, C. DuPuis, and A. Sheretov, A Methodology for Testing Spreadsheets, *ACM Transactions on Software Engineering and Methodology,* (to appear).
17. K. Rothermel, C. Cook, M. Burnett, J. Schonfeld, T. Green, and G. Rothermel, WYSIWYT Testing in the Spreadsheet Paradigm: An Empirical Evaluation, *International Conference on Software Engineering* (June 2000).
18. B. Shneiderman, Direct Manipulation: A Step Beyond Programming Languages. *Computer* **Vol.16, No.8** (Aug. 1983) 57-69.
19. D. Smith, A. Cypher, and J. Spohrer, Kidsim: Programming Agents without a Programming Language. *Communications of the ACM* **Vol.37, No.7**, (July 1994) 54-67.
20. J. Stasko, A. Badre, and C. Lewis, Do Algorithm Animations Assist Learning? An Empirical Study and Analysis, *INTERCHI '93* (Apr. 24-29, 1993) 61-66.
21. Tanimoto, S., VIVA: A Visual Language for Image Processing, *Journal of Visual Languages Computing* **Vol.2, No.2** (June 1990) 127-139.
22. T. Teo and M. Tan, Quantitative and Qualitative Errors in Spreadsheet Development, *Hawaii International Conference on System Sciences* (Jan. 1997) 149-155.
23. R. Walpole and M. Burnett, Supporting Reuse of Evolving Visual Code, *1997 IEEE Symposium on Visual Languages* (Sept. 23-26, 1997) 68-75.
24. K. Whitley, Visual Programming Languages and the Empirical Evidence For and Against, *Journal of Visual Languages and Computing* **Vol. 8, No.1** (Feb. 1997) 109-142.
25. E. Wilcox, J. Atwood, M. Burnett, J. J. Cadiz, and C. Cook, Does Continuous Visual Feedback Aid Debugging in Direct-Manipulation Programming Systems? *ACM Conference on Human Factors in Computing Systems (CHI'97)*, (Mar. 22-27, 1997) 258-265.