View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UCT Computer Science Research Document Archive

Holistic Programming Environments

Gary Marsden[a] Harold Thimbleby[b]

[a]*Department of Computer Scince, University of Cape Town, South Africa,* gaz@cs.uct.ac.za
[b]*School of Computer Science, Middlesex University, London, UK,* harold@mdx.ac.uk

Abstract

As a result of the popularity of graphical user interfaces, it is now almost impossible to buy a programming language compiler – instead, one purchases a development environment. Of course, we can scoff at the distinction and say that a development environment is nothing more than a programming language with visual (as opposed to syntactic) sugar.
We believe, however, that this view must change if safer and more responsible programming languages are to be created for the next generation of programmer. Within this paper, we will argue that a more theoretical approach should be taken to the development of programming environments and suggest ways in which this may be achieved.

Keywords: *Programming languages, GUI programming, Interface design*
Computing Review Categories: *D.1.5; D.1.7, D.2.6, H.1.2*

1 Development environments

In the early days of graphic user interfaces, toolkits (or API's) were created which allowed programmers to access code libraries that could be used to create basic interface elements (widgets). The success of early toolkits such as X-Motif[2] was immense, and it has been estimated that these toolkits provided a ten-fold increase in the productivity of interface programmers[8]. However, the real power of these toolkits was only realised when they were coupled to an interface development environment. Rather than having to set visual attributes (such as position and colour) using textual commands, the visual environment allowed programmers to use direct manipulation of widgets to set these visual properties.

Consequently, current development environments are usually comprised of three main elements: a language compiler, a visual toolkit and an interactive environment (usually with a text and visual editor). Currently, each of these components is developed separately by the manufacturer. This may be on purpose (to reduce costs as with Microsoft's Visual Studio set of tools) or it represents a throwback to historical development strategies, where the editor was supplied independently of the compiler. Whilst tools in current development environments are integrated at some level (e.g. the editor knows how to colour the syntax of the source code) the development of the environment and toolkit is still largely separate from the development of the language.

We believe that the power of development environments can be greatly improved through the deliberate integration of editor, language and toolkit.

This idea is not new, and the importance of integrating a language to its environment has been realised in projects such as Oberon[10] and Smalltalk[3]. When these systems were released, however, it was unusual to have a language supplied with its own environment. Also, the processing demands of such a system were greater than those which were readily accessible to most programmers. This is no longer the case. It is therefore time to look at the problems caused by maintaining the separation between language and environment and also to investigate the potential benefits of integrating them.

2 State of play

Visual Basic[6] is a typical example of a third generation programming language supported by a visual environment. As such, we shall use it as an exemplar to investigate the issues arising from combining a text based language with a visual environment. Although the comments in this section are specific to Visual Basic, the same questions can be applied to any other programming language and environment.

The provision of a widget toolkit and a graphical editor greatly increases the utility of BASIC. Programmers can use these facilities to rapidly develop new applications which would have been impossible before. Like every other language, however, BASIC has it weaknesses, which are well understood and documented[1]. However, by adding the toolkit and the environment, have we have removed some weaknesses but have we added some more? Consider the following example:
A variant variable is used within the Visual Basic programming language to hold data of any type. Similarly, a text box is a widget which can be used to hold data of any type. Imagine a program with two variant variables (`vv1` and `vv2`) and two text boxes (`tb1` and `tb2`). Into all the text boxes and variables we insert the value five. Perform-

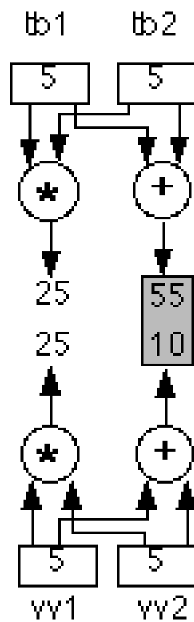

Figure 1: Inconsistent coercion in Visual Basic

ing the calculation `vv1 * vv2` gives the same result as `tb1 * tb2`, namely 25. Performing the calculation *vv1 + vv2* gives the result 10. The calculation `tb1 + tb2` results in the answer 55! Text boxes are coerced to a string for the '+' operator, whilst variant variables are coerced to integers. Whether or not you regard automatic type coercion as a good thing, one might at least expect that the coercion is performed identically for semantically identical concepts! (Of course, the problem really lies with the overloading of the '+' operator and could be solved by using different symbols for concatenation and addition.) This process is sumarised in Figure 1.

At this point, we can throw up our hands in indignation and blame the Visual Basic programmers for their incompetence. However, this does little to further computer science and it is worth considering how other errors of this type could be spotted.

One way to do improve visual language design is to change our attitude to the "visual"part of the language. Rather than treat it as some sort of add-on, we could undertake to examine the semantics of toolkits and editors and see how those relate to the semantics of the underlying language. By treating the environment as being as important as the language it supports, we can perhaps shed some light on how to design more effective environments which compliment the language more appropriately. Furthermore, by investigating the environment's semantics, we can go on to use studies in third generation languages which have given us well founded design principles – something which does not yet exists for language environments. These design principles from text based languages turn out to be just as applicable to the environment which encompases the language. We shall investigate how some of these principles could be used to create more responsible environments.

2.1 Principle of Correspondence

The principle of correspondence[9] when applied to programming languages, enables the programmer to treat semantically identical items in a syntactically identical manner, without having to be directly concerned with small, inessential details. The principle could be extended to apply to semantically identical items in the language, toolkit or environment.

Returning to the problem outlined above, we can see that text boxes are semantically equivalent to variant variables. Therefore, the principle of correspondence would have told us that values held in text boxes and variant variables should be treated in the same way. The fact that text box values are visible to the user, and variant values are not, should have no impact on how those values are used in calculations.

2.2 Principle of Generality

The principle of generality[5] states that there should be no special cases within a language. Setting aside any inconsistencies within BASIC, there are special rules to remember when accessing the toolkit from the Visual Basic language or the Visual Basic environment. Most common among these is setting attributes of widgets – some attributes can only be set in the environment and not from the language. This might have made sense had the attributes only been accessible at run-time, but for some widgets, certain attributes are never accessible from the language. Clearly, by considering language and environment together, this type of error could have been spotted at the design stage.

2.3 Universe of discourse

Traditionally the term "Universe of discourse"[7] refers to the different data types a language can process. General purpose languages, therefore, included types permitting the processing of data held by the host operating system and machine hardware. At the time when most third generation languages were designed, this would include strings, characters and numbers.

Considering the integration of toolkit to language, it would seem sensible to extend languages so that they could include bitmaps and sounds as first class data types. Without the inclusion of these types, the universe of discourse is incomplete. Few languages treat multimedia types as first class.

It is interesting to make a comparison of the difference between the universe of discourse for the language and the universe of discourse for the graphical environment of a given environment. For example, the interface builder in Visual Basic is able to create and destroy widgets; but this is not possible to achieve using Visual Basic programming constructs. Furthermore, there are some development environments which use widgets in the environment which are not available in the language. This results in an internal inconsistency, presenting a tool which appears to provide a

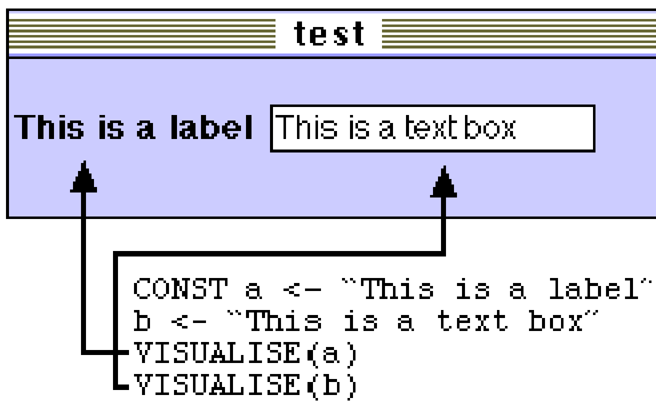

Figure 2: Fields and labels visualised from variables and constants

certain functionality, yet that functionality is not available from the programming language.

One way the universe of discourse (and indeed other attributes) of third generation languages was tested was to implement the language compiler in the language itself (meta-circularity). This became a fairly standard test and languages which did not pass it were, in the words of Levey[4] "beneath contempt."

The equivalent check for a visual development tool would then be to implement its interface builder using the language component of the tool. This would not only ensure an appropriate universe of discourse, but it would make the tool more flexible and freely customisable to those wishing to change it. This would also be a conceptually optimal way to integrate the language and environment.

3 Integration

In the previous section, it was shown that problems within current development environments could be spotted using well understood principles, provided the environment was considered to be part of the language. The problem then remains of bringing the semantic concepts of the environment into the language, allowing this integration to take place. In order to determine if such integration is possible, the semantics and roles of common interface widgets were examined and suggestions made as to how they could be represented in a programming language. A selection of the target widgets is presented beneath.

3.1 Fields and labels

Perhaps the simplest place to begin is with text boxes and labels as these are essentially strings displayed on the interface – assuming that the language has string types and string constants. The *presented* version of a string constant is a label and the *presented* version of a string variable is a text box. (See Figure 2).

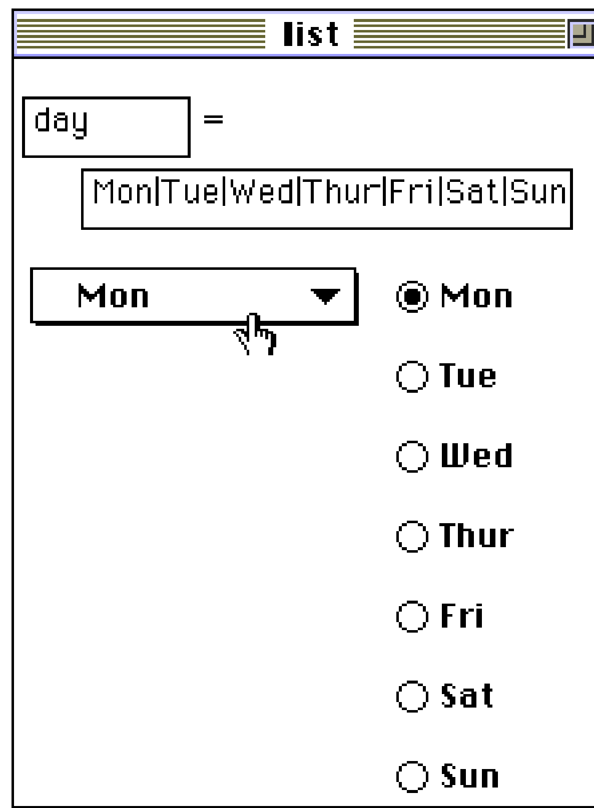

Figure 3: Fields and labels visualised from variables and constants

3.2 Radio buttons

Radio buttons, like text boxes, already have a directly equivalent concept within programming languages, namely the enumerated type. Variables of a particular enumerated type can only hold one value from the set declared in the type definition. For example, in Pascal, the type day can be declared as:

```
TYPE day =(Mon,Tue,Wed,Thur,Fri,Sat,Sun);
```

Obviously, as radio buttons provide a mutually exclusive choice, they are semantically equivalent to the enumerated type. Furthermore, enumerated types are also semantically equivalent to pop-up menus, permitting a choice of visualisations as in Figure 3.

3.3 Check boxes

The check box is a close relative of the radio button, but provides selection from a non-exclusive list. When the list contains only one item, then the check box becomes a visualisation of a Boolean variable. If the list is longer, then it becomes necessary to introduce a new concept which we shall term the *Variant Enumerated* (VE) type. Similar to the enumerated variables, variables declared of VE type can hold a list of values taken from the defined set, rather than the single value of the enumerated variable. For example, to implement a variable to hold the style of text in a word processor, the variable could take one or more values from the list (bold, italic, underline), as in Figure 4.

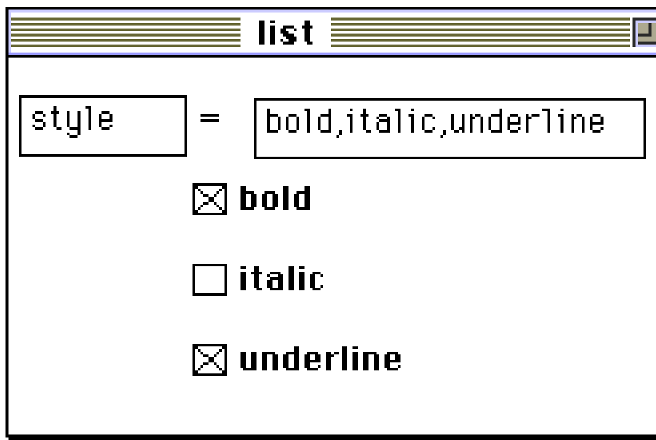

Figure 4: Style menu showing inclusive choice

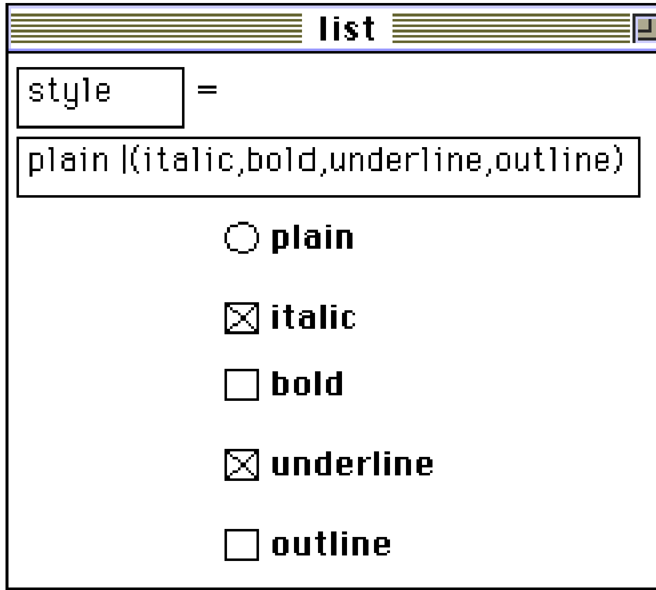

Figure 5: Exclusive and inclusive choices

Whilst it so happens that these values are compatible, it can also be the case that selection of a particular value excludes other values. Again, taking font style as an example, the value "plain" would exclude other values. The declaration of a VE variable should therefore permit the exclusivity and inclusivity between values. The notation adopted uses boolean connectors, thus making the declaration of this style menu:

```
style = ( plain | (
italic,bold,underline,outline ) );
```

In this notation, the vertical bar is used to represent an exclusive conjunction, and the comma used to represent an inclusive conjunction. This can then be presented as in Figure 5.

3.4 Extending the process

This process can be continued to the point where most widgets can be represented as data types in a programming language which would be familiar to programmers working with current development environments. In fact, it is not even necessary to create an entirely new language, as the flexibility of an OO language like Java allows us to create a class of visual variables to replace the standard types of "Integer", "String" etc. Varaibles from the visual varaible class could have a "display" method which caused the variable to be rendered as part of the interface. Subsequent changes in the value of the variable are then automatically reflected in the interface. Such a scheme was implemented as a trial in Java. (See Figure 6).

4 Benefits

If language-environment integration was pursued in way similar to that presented above, we believe it would create a number of benefits, beyond the removal of errors

- *Improved interface design*: By encoding the semantics of a particular interface widget within the language, the programmer is forced to think about widgets at a higher level. This makes the purpose of each widget more apparent to the programmer and should allow them to choose the most appropriate representation for any given concept.

- *Improved learnability*: By removing the distinction between language and environment, the programmer need only learn one set of concepts; not the different abstractions found in separate toolkit and language.

- *Programmer support*: Formerly, languages such as C++ required that the programmer understand some advanced concepts (e.g. pointers and object hierarchies) before they could create even the simplest interfaces. By making the interface a visualisation of the concepts already learnt as part of the language, the programmer can create interfaces for programs written at the level of their proficiency.

- *Consistency of GUI*: By allowing the system to select widgets, more coherent interfaces will be produced. This can be further improved by customising the system to adhere to a desired interface style guide.

5 Conclusions

We have presented how treating a language and its environment as separate entities can lead to problems when using the two elements together. We have shown that these problems can be removed, and other benefits created, by blurring the distinction between language and environment. In an effort to show how this blurring might occur, we suggested how a small subset of widgets could be more closely integrated with a programming language. If this work is to be completed, however, then it will be necessary to show how more widgets can be integrated into a language.

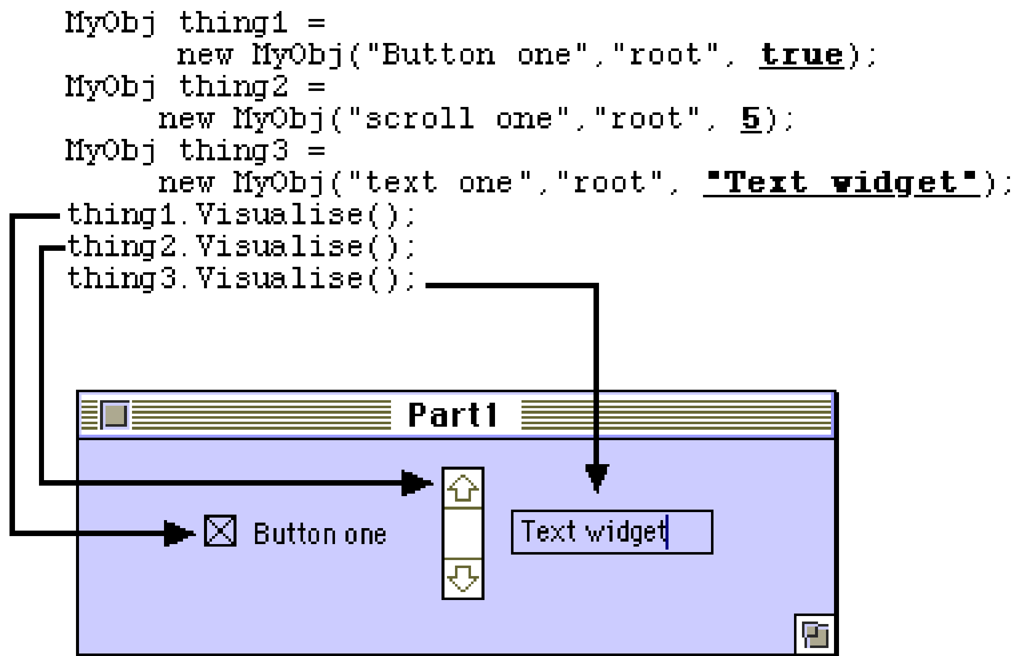

Figure 6: Sample Java implementation

References

[1] E. Dijkstra. *Selected Writings in Computing*, chapter How do we tell truths that might hurt? Springer Verlag, 1975.

[2] Open Software Foundation. *OSF/Motif Programmer's Reference Release 1.1*. Prentice Hall, 1990.

[3] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[4] S. Levey. *Hackers*. Penguin, 1984.

[5] K. Louden. *Programming Languages: Principles and Practice*. PWS-Kent, 1993.

[6] Microsoft. Visual basic product description. Technical report, www.Microsoft.com/vbasic/, Last visited July 2000.

[7] R. Morrison. Towards simpler programming languages: S-algol. Technical report, St. Andrews University, 1983.

[8] B. Myers. *Languages for developing User Interfaces*, chapter 1. Jones and Bartlett, 1994.

[9] R.D. Tennent. Language design methods based on semantic principles. *Acta Informatica*, 8:337–349, 1977.

[10] N. Wirth. The programming language oberon. *Software - Practice and Experience*, 18(7):671–690, 1988.