

Software development by continuous gesture

A discussion paper

Luke Church
luke@church.name

ABSTRACT

Existing development environments are almost exclusively keyboard driven making them unsuitable for use by developers who suffer from motor difficulties such as RSI. This discussion paper introduces the possibility of using a continuous gesture system and language model to provide an alternative entry mechanism and looks at some of the questions raised by the design of such a system.

1 INTRODUCTION

Most current popular software development systems are keyboard driven, whilst this provides a familiar mechanism for code entry it creates problems for developers with motor difficulties, including repetitive strain injury and related disabilities

Dasher[1] allows for natural language text entry for people with mobility limitations as well as devices with limited data entry facilities such as Pocket PCs. However its language model makes it unsuitable for software development. This discussion paper looks at the possibility of building a new language model and interface based on the Dasher system that would enable software development to be completed by continuous gesture.

This would enable developers with motor difficulties to work more effectively and would provide an alternative option to typing that would help to delay the onset of RSI and related problems. There are also possible benefits associated with the mode of entry that might assist new developers and increase the accuracy and speed of experienced developers.

2 CONTINUOUS GESTURE FOR SOFTWARE DEVELOPMENT

2.1 Dasher as a continuous gesture system

The Dasher system operates by the user continuously motioning towards the next character that they wish to input. So to write the word 'The' the user would point the cursor to the rectangle representing 'T'. Vertically aligned within the rectangle representing 'T' are rectangles representing the various combinations that are possible. They appear in alphabetical order, 'Ta', 'Tb' etc. the user then motions towards the rectangle representing 'Th', this is larger as it has a higher probability of occurrence in English.

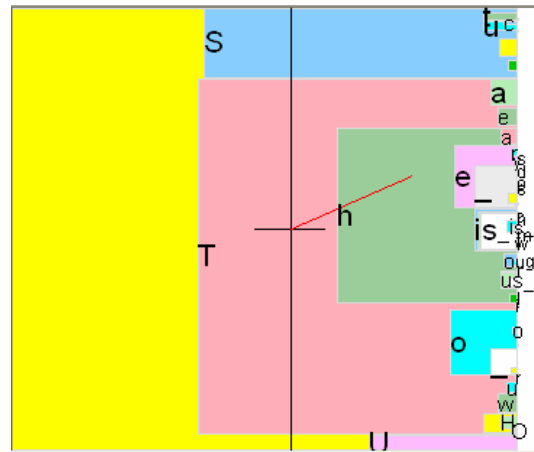


Figure 1

Figure 1 shows the state of the display after 'T' has been selected and as the user guides to the cursor towards 'h' and then 'e'.

An animated demonstration of Dasher is available on the Internet [2]

In continuous use this gives the impression that the user is zooming or steering towards the input they desire.

Correction is possible by positioning the mouse cursor to the left of the central line and 'steering backwards'. A Dasher expert has achieved an average speed of 170 characters per minute, there is some evidence that the accuracy is higher than the accuracy commonly achieved using a keyboard [1]. Dasher has been used for text entry by eye tracking, by breath and running on a Pocket PC.

Dasher operates using a probabilistic language model based on PPM which gives an entropy of some two bits per character.

2.2 Development with continuous gestures

Object orientated frameworks have a natural hierarchy that intuitively translates into a Dasher style system. The constrained range of possibilities at any given point increases the suitability of an object framework for a Dasher style input system.

Such an object orientated input hierarchy can be viewed as a form of tree whereby each choice constrains the available options to the branches of that choice. Guiding the cursor to a particular rectangle is equivalent to selecting a branch of the tree.

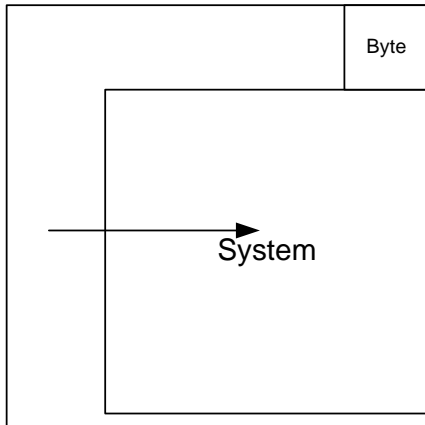


Figure 2

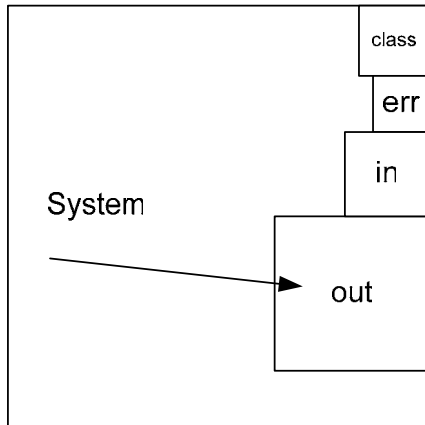


Figure 3

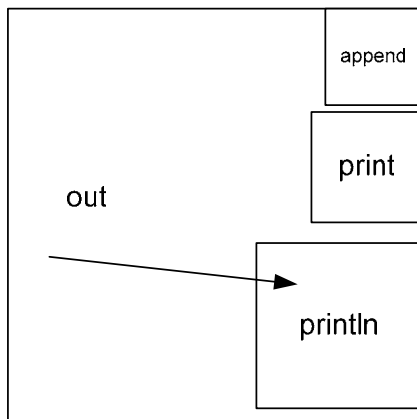


Figure 4

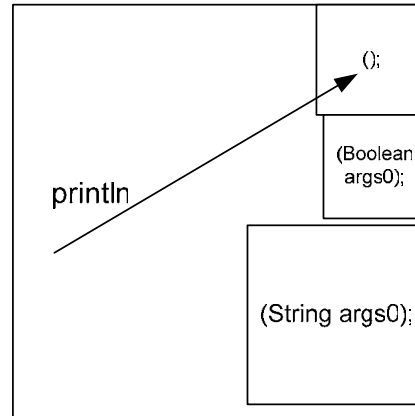


Figure 5

Figures 2 to 5 show a simplified example of how one might input statements such as 'System.out.println();' it should be noted that this is a simplified example, it excludes the full library of choices and excludes the other possibilities such as navigation. It is a graphical impression rather than a capture from a live system, the probabilities and hence sizings are arbitrary.

First in Figure 2 the user selects 'System' from the default scope of 'java.lang' then in Figure 3 'System.out', in Figure 4 the method 'System.out.println' is selected and finally in Figure 5 the empty overload is selected.

Such a development system would actively support encapsulation and data hiding by only offering objects that are in scope at a given point in the code and only offering methods and attributes that are available against those objects in the current context.

While such a system is useful for ensuring that syntactically correct code is entered there will be circumstances when the developer needs to enter arbitrary text, such as when manipulating objects for which the syntax has not yet been inputted/imported or when creating comments/code documentation lines or dialog texts. Similarly, much of a developer's time is spent navigating and editing pre-existing code rather than creating new code from scratch.

To accommodate these three modes of operation (tokenised validated entry, arbitrary text entry and navigation) the development system is proposed to have three top-level blocks, each associated with one of these activities, in a similar manner to the way Dasher has multiple blocks for lowercase entry, uppercase entry and special characters.

The code navigation block is currently proposed to offer a hierarchical view through the code, first navigating through package level, then class level and block level etc.

the details of the system that will be used for navigation is currently an area of research.

2.3 Advantages of continuous gesture systems for development

Continuous gesture systems offer a number of advantages for development environments. The combination of encapsulation derived restrictions and the increase in accuracy associated with the input mechanism, as seen in Dasher[1], may result in a lower rate of defects that are artefacts of typing or spelling errors, this would be valuable to overall developer productivity.

The system provides a self consistent approach to code entry and code navigation in that both are viewed as exploring an object hierarchy. The system also offers the possibility of presenting a larger amount of information at input time enabling additional metadata to be displayed in realtime about the decisions the developer is making.

Further, the use of alternative input mechanisms has already been demonstrated with Dasher, these mechanisms should allow partially disabled and disabled users to continue to be productive in software development.

3 PROBABILISTIC LANGUAGE MODELING FOR JAVA

3.1 The need for probabilistic language models

A Dasher style input system depends on a probabilistic language model to size the blocks from which the user selects. To support the object orientation development paradigm the language model must also constrain the possible selections to those options that are legal in the given context.

The navigation section of the language model must provide a probabilistic view of the likelihood that the developer wishes to edit the code within that particular block Of the.

A further requirement on the model is that it must be sufficiently high-performance to be able to return a suitable number of probabilities to the graphical interface in realtime.

3.2 Context sensitivity and language the awareness

The Dasher system gains its efficiency and usability improvements over arbitrary text entry from its language model.

The requirements of the language model for an object orientated language are more complex than those for natural language, a symbol by symbol probabilistic compression system such as PPM is not suitable for an object orientated language which has more complex and strictly defined rules. A language model that takes these

rules into account should be able to achieve a higher efficiency than a simple symbol by symbol representation as well as aligning more consistently with the way a developer thinks about code.

Without such a validating language model the system would become actively confusing as it would, for example, suggest methods there are valid in a different scope but are invalid in the current scope, this would be misleading to the developer especially with commonly used variable names.

Hence there is a requirement for the language model to have an awareness of both the framework that ships with the language and the languages' keywords.

3.3 Self updating models

The model of the language must adapt in realtime to take newly added code into account. As code is added or as the user navigates through the code the current scope and hence the legal possibilities will change. The model must keep in step with these changes in an efficient and timely manner.

The model must also adapt to changes in code such as changes to access modifiers that can affect the legality of other parts of the code

3.4 Probabilistic modeling through large-scale, weighted learning

The intention is to apply the Dasher paradigm that adaptivity and learning are preferable to preprogramming. The intention is that the model should be trained against a large body of code that uses the framework. It may be necessary to preprogram some parameters and rules about the language's keywords and structure. As mentioned previously the model will be required to self adapt. It is likely that the classes used by the current project are more likely to be used again than some arbitrary class in the framework hence it should be considered in the probability generation that the current project is more important than the pre-learnt history.

The balance of importance between historically learnt code and the current project is a current research question.

4 ENTROPY MODELING OF COMPUTER LANGUAGES

An entropy analysis of an object orientated language would allow bounds to be placed on the limits on the efficiency that could be expected from the code entry system.

The entropy of an object orientated language and its associated framework will be low, this can be seen from the manner in which the language can be tokenised and predicted. In a very high entropy language such prediction

would not be possible. Another indication that the code has low entropy is that it is highly losslessly compressible.

An analysis of the entropy of an object orientated language is an ongoing research task.

5 ENTRY SYSTEM RESEARCH

The Dasher system displays any items with very low probability with a fixed small probability, δ , to enable a user to find items of low probability more easily. This is inconsistent with the object orientation paradigm of encapsulation and hence the facility to include probabilities that literally equal zero seems to be a necessary addition.

How much code should be automatically added? For example, should the system introduce brace pairs or only the opening brace when a block is started? If only the opening brace is inserted should the system recompute all subsequent code in the new incomplete scope?

How much template code should be added?

How should the system handle the file and project management requirements of the language? Should the system operate in files or projects?

5.1 Visual clues

How much information should be presented to the user? Can adequate information be presented within the blocks for the user to select the correct method overload? How large, and hence how probable, should the blocks be before the system starts populating them with this information?

Should other metadata such as Javadoc summary information be presented?

What is the optimal syntax highlighting technique to use and what colours should be reserved by the system for highlighting the different blocks? (token entry, arbitrary entry, navigation)

What is the best way to order such information? Dasher depends on the user knowing the alphabetical ordering of the language, are there better orderings available for coding? How successful will a user be at locating code within a large class? Could the system be adapted for a minimum motion requirement?

Dasher requires continuous visual attention, how can this be reconciled with the requirement to see large blocks of code?

5.2 Code navigation structure

How do we elegantly add state to a language model? How do we consistently add 'jump to function definition' type commands to hierarchical navigation structure?

Is a hierarchical navigation structure adequate for navigating through large projects?

How can diagramatics such as UML be added to such a system?

5.3 Error tolerance

The ability to add arbitrary text results in the possibility of adding syntactically and semantically incorrect code.

How should the user be alerted of such errors?

6 LANGUAGE MODELING RESEARCH

6.1 Entropy modeling

What tokenisation is appropriate when minimising code for the purposes of entropy analysis?

Taking the tree model of code outlined earlier, what is the branching factor?

Can the use of code modelling techniques such as patterns or micro-patterns be used to further increase efficiency and accuracy?

6.2 Error tolerant structural parsing

How should errors be handled within the structural parsing system?

How should processing continue?

How should the system deduce which are references to blocks of code yet to be added and which are genuine errors?

How can the system undo changes to the structural model without full recomputation?

How should the system handle pathological cases such that the system doesn't become inoperable?

6.3 Optimal probability distribution

How important is the learnt history associated with the framework versus the learnt history associated with the current project versus the usage in the current scope?

6.4 Optimal tokenisation

What is the optimal tokenisation of object orientated code? For example should the braces be included in method declaration tokens or do they represent a separate distinct token?

REFERENCES

1. D. J. Ward *Adaptive Computer Interfaces* PhD thesis, Cambridge University UK, 1999
2. Demonstrations of Dasher
<http://www.inference.phy.cam.ac.uk/dasher/Demonstrations.html>