SketchCode – An Extensible Code Editor for Crafting Software

Siemen Baader(✉) and Susanne Bødker

Department of Computer Science, Aarhus University,
Aabogade 34, 8200 Aarhus N, Denmark
{sb,bodker}@cs.au.dk

Abstract. We present SketchCode, a code editor that its users can augment with visual elements to represent domain and program concepts. We examine programming as sketching and identify the techniques of *postsyntactic augmentation*, *macro components*, and *interactive semantic enrichment*. Based on studies of programmers, we discuss these techniques as a promising way for code editing and tool appropriation.

Keywords: Programming · Crafting · Sketching

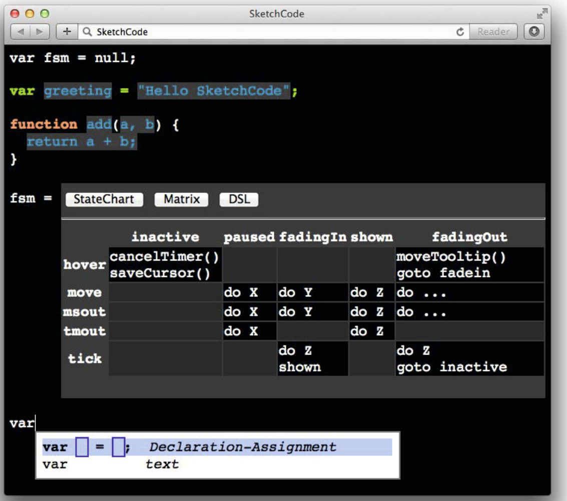
1 Introduction

Domain specific programming tools can be very effective at supporting development and workflows within their particular area. We note that programmers often act as end-user developers in the sense of Ko et al. [2], in that they create and appropriate tools to fit their domain and own ways of working. Similarily, designers make extensive use of sketching and model making as ways of expressing a design in the making. Lindell characterized both disciplines as 'crafting' [4], and we believe that programming and designing share similarities in their relieance on thinking through action and the creation of supporting structures as central activities. While the creative use of expressive materials is widely accepted within design professions, programmers do not have access to modifying their code writing tools in the same practical ways, and most end-user appropriation of programming tools is not very feasible beyond the configuration of standard editors and refactoring of source code. We report on early findings from a design-oriented effort to understand the end-user development aspects of programming. We ask *"What can we learn from design studies to support the expressive and appropriative aspects of programming practice, and how can we design programming tools to support this?"*

In this paper we report on two things: How end-user development in programming can be interpreted as design, in particular sketching, and which requirements this interpretation offers. Based on an empirical study of programmers we present SketchCode. We introduce the interface techniques of *postsyntactic augmentation*, which allows the inlining of rich visual editors in plain text code,

P. Díaz et al. (Eds.): IS-EUD 2015, LNCS 9083, pp. 211–216, 2015.
DOI: 10.1007/978-3-319-18425-8_18

macro components, which are rich interactive editors for domain and program concepts, and *interactive semantic enrichment*, which allows the insertion of non-parseable parts into source code. Figure 1 shows how source code is augmented and intermixed with components that represent higher level concepts.



```
var fsm = null;

var greeting = "Hello SketchCode";

function add(a, b) {
  return a + b;
}

fsm =
```

	inactive	paused	fadingIn	shown	fadingOut
hover	cancelTimer() saveCursor()				moveTooltip() goto fadein
move		do X	do Y	do Z	do ...
msout		do X	do Y	do Z	do ...
tmout		do X		do Z	
tick			do Z shown		do Z goto inactive

```
var
  var [] = [];   Declaration-Assignment
  var            text
```

Fig. 1. The SketchCode extensible editor displays the concept of *postsyntactic augmentation*. It shows white plain text source code (top line), colored *macro components* of varying complexity, and *interactive semantic enrichment* via an autocompletion menu.

2 Programming seen from a Sketching Perspective

To identify theoretical requirements for a programming system informed by design studies, we rely in the concepts of crafting, reflective practice and sketching. Lindell [4] demonstrated that interaction designers and programmers share a crafting epistemology. Likewise informed by reflective practice, Buxton [1] and Lim et al. [3] studied the specifics of sketching techniques and prototypes in design. Applying these perspectives to programming results in the following perspectives on code and editors as a design material:

Backtalk. Reflective practice is a dialectic process and depends on backtalk from the code and editor in order to advance the solution (reflection in action) or to rephrase the approach (reflection on action). In programming, we have observed backtalk from four sources: running the code, reasoning about the code, feedback from static analysis, and representing the code differently (e.g. in a diagram). To support the cycle of reflective practice, a system can e.g. tolerate broken and pseudo code, simultanously allow different (e.g. visual) representations and perform automated reasoning.

Externalizing and Improvising. Lim et al. [3] introduced the notions of *filtering dimensions* and *manifestation dimensions* as ways to using protypes and sketches economically. In coding, filtering dimensions include properties like execution order, stateful situations like in user interfaces and visual properties like color codes. Effective manifestations include meaningful names and refactored code, but also graphical representations like charts, tables and formulae available in a modern browser.

Sketching and Modeling. Visualizations and automated reasoning approximate the agenda of modeling tools. Unlike offline sketches, models and representations in code can be reused and linked with the concepts they represent. However, modeling tools require the complete specification of models to work, and in order not to prevent other kinds of backtalk, a system informed by sketching should allow, but not require, complete modeling.

3 Programmer Studies

We conducted 6 participant observations of 30 minutes and semi-structured interviews with web-programmers in two startup companies. In addition we collected 103 scenarios of interest to the sketching perspective in the form of annotated screen shots from the first author's own programming practice.

Backtalk. The most direct mode of backtalk was making a change and executing the program to see the result. Programmers furthermore relied on mental execution to find bugs, e.g. reasoning about front-end code that was broken beyond execution. Static analysis was another source of backtalk, e.g. using the JSLint static analysis tool regularly without executing the code. A final source of backtalk comes from the representation, e.g. using state charts and transition matrices to reason about complex network interactions. We observed backtalk from a variety of sources, and rather than siding either with dynamic (backtalk from execution) or compiled languages (backtalk from static analysis), programming systems should provide backtalk suitable to the specific situation.

Representation. Several situations indicated that the programmers benefited from concise representations of the programing concepts at hand, e.g. preferring the concise JQuery library over the browser's verbose DOM API, or writing hexadecimal CSS color codes and repeatedly refreshing the browser in trial-and-error cycles to find a desired color.

Modeling. The situations of representing colors and the different state chart representations lend themselves not only to visual representation, but also to direct manipulation interfaces and concept-specific editing assistance. The issues of representation and receiving relevant backtalk go hand in hand with issues of modeling, i.e. elevating the source codes semantic level beyond the textual level.

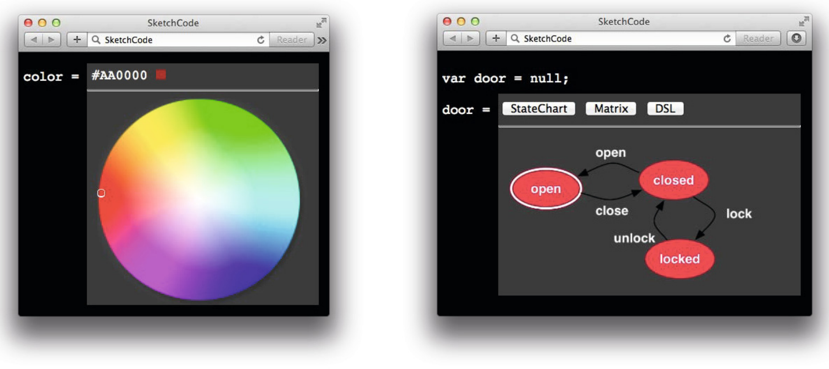
Emergence and refining. The programmers saw static programming systems as heavyweight and dominated by boilerplate, which indicates that a system should be vary of imposing formal structures up front, and allow the coexistence of higher level concepts, visual representations and very crude code, and the gradual refinement of code into higher level components. One programmer e.g.

used Emacs because it allowed him to execute code in an anonymous buffer without having to name a file.

4 The SketchCode Extensible Code Editor

Figure 1 shows the SketchCode editor. It is implemented as a number of partly usable prototypes used to develop the conceptual design. The editor contains both source code in plain text (top line) and instances of *macro components* representing the declaration and assignment of a variable, a function, and one representing the configuration of a finite state machine using three panels (a state chart, a transition matrix and code). The programmer writes code and navigates the editor like a standard editor. However, macro components govern their own user interfaces and support and restrict editing according to their meaning. Since macro components can represent both very complex but also very simple concepts in code, the editor does not offer syntax coloring but instead provides macro components for basic concepts such as functions and variables.

The user inserts instances of macro components into the source code using an autocompletion menu, which we call *interactive semantic enrichment*. In Figure 1, the user is just about to insert another instance of a variable declaration and assignment component. Editors within macro components may recursively contain other macro components. At run time, the macro components are expanded to valid syntax within their surrounding region. E.g., a macro component representing a CSS color expands to the string `#CC0000` within a CSS stylesheet, while it will expand to `"#CC0000"` to be a valid expression in a JavaScript context.



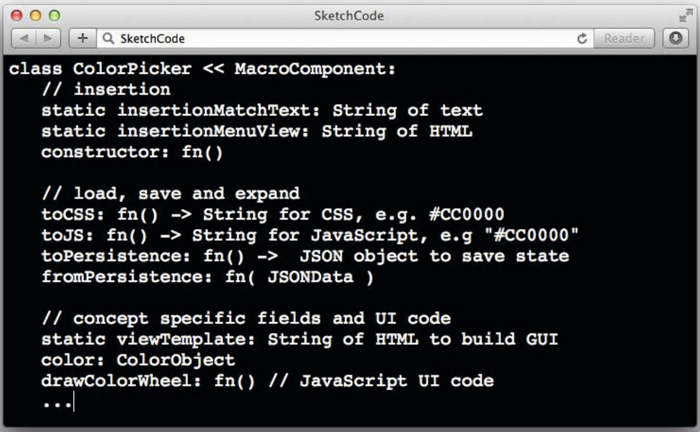
```
color = #AA0000
```

```
var door = null;
door =
```

Fig. 2. Visual Macro Components: A CSS color wheel (left) and a state chart (right)

Macro components are built as small interactive web user interfaces, and integrate with the surrounding SketchCode environment in a standardized way. Figure 3 shows pseudo code for an interactive color picker to edit CSS color codes. Complex macro components such as the state machine contain executable code in their expanded representation, and to avoid name clashes, this should be implemented as hygienic macros.

The key aspects of this design consist of three parts. First, it allows the gradual refinement of plain text code into more semantic representations only as needed – the use of macro components is not enforced, and the editor can be used as a plain code editor until specialized macro components are needed. Second, the creation of macro components makes use of the skills that the programmers are proficient in. Third, the system integrates with the text-based ecosystems that programmers live in, like version control and interpreters.



```
class ColorPicker << MacroComponent:
   // insertion
   static insertionMatchText: String of text
   static insertionMenuView: String of HTML
   constructor: fn()

   // load, save and expand
   toCSS: fn() -> String for CSS, e.g. #CC0000
   toJS: fn() -> String for JavaScript, e.g "#CC0000"
   toPersistence: fn() ->  JSON object to save state
   fromPersistence: fn( JSONData )

   // concept specific fields and UI code
   static viewTemplate: String of HTML to build GUI
   color: ColorObject
   drawColorWheel: fn() // JavaScript UI code
   ...
```

Fig. 3. The macro component programming interface shown in pseudo code

We conducted an early evaluation of the SketchCode concept by confronting the programmers with the prototype and discussing how they would use and extend it. Reactions ranged from excitement to skepticism with regard to the efficiency and uniform editing capabilities achieved in text-only environments. One sceptic noted that “everything can be expressed in text, and I believe it should”. On the other end, one programmer noted that this was “a very interesting concept, I already build visualisations for my database.” Two of them have since started implementing visual code editing in their own toolmaking.

We draw two main conclusions. First, lightweight tooling is important to the audience, and a running prototype must test if the system remains efficient and easy to use with an increasing number of macro components. Second, we have seen evidence of toolmaking and appropriation, but programmers are not fully aware of the effects of changing and evolving representations and should be educated in this way of thinking before the concept can be tested properly.

5 Discussion and Conclusion

We have introduced the perspectives of crafting and in particular sketching, identified central elements of it in programming practice and proposed the design of SketchCode and its key concepts of *postsyntactic augmentation*, *macro components* and *interactive semantic enrichment* to accomodate sketching better in code editors. With this, we argue for a more fine-grained view on the process of programming, its different kinds of backtalk, and how it can benefit from evolutionary creation of supporting semantic structures and visualizations.

Language workbenches address issues of backtalk from static analysis and representations well, but they require the programmer to work directly on the abstract syntax tree. This requires programmers to re-learn their editing tools and to define concepts up-front [6]. We propose a more gradual approach, where most code can be edited in a traditional way, and semantic editing introduced gradually as needed. Agentsheets [5] is another toolkit for creating and evolving (visual) programming languages, and differs mainly from SketchCode in that the latter uses plain text and web technology concepts from within the domain of the end users. Live programming systems such as Processing and Smalltalk cater more to getting backtalk from executing code than SketchCode, and less to backtalk from representation.

Overall, SketchCode presents a balance towards supporting backtalk from representation and domain specific editing support, while taking into account the needs for emergence, existing practices and known technologies within the domain of its user population.

Acknowledgments. We thank Joseph Kiniry for exceptional support, and Clemens Klokmose and our three reviewers for constructive feedback.

References

1. Buxton, B.: Sketching user experiences: getting the design right and the right design. Morgan Kaufmann (2010)
2. Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., et al.: The state of the art in end-user software engineering. ACM Computing Surveys (CSUR) **43**(3), 21 (2011)
3. Lim, Y.K., Stolterman, E., Tenenberg, J.: The anatomy of prototypes: Prototypes as filters, prototypes as manifestations of design ideas. ACM Transactions on Computer-Human Interaction (TOCHI) **15**(2), 7 (2008)
4. Lindell, R.: Crafting interaction: The epistemology of modern programming. Personal and ubiquitous computing **18**(3), 613–624 (2014)
5. Repenning, A., Sumner, T.: Agentsheets: A medium for creating domain-oriented visual languages. Computer **28**(3), 17–25 (1995)
6. Voelter, M., Siegmund, J., Berger, T., Kolb, B.: Towards user-friendly projectional editors. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) SLE 2014. LNCS, vol. 8706, pp. 41–61. Springer, Heidelberg (2014)