A Toolkit Approach to Sketched Diagram Recognition

Beryl Plimmer
Department of Computer Science
University of Auckland
Auckland, New Zealand
+64 9 373 7599
beryl@cs.auckland.ac.nz

Isaac Freeman
Dept of Computer Science & Software Engineering
University of Canterbury
Christchurch, New Zealand
+64 21 1511209
ijf23@student.canterbury.ac.nz

ABSTRACT

Sketch-based tools provide a more human centered design environment than traditional widget-based computer design software. A number of sketch tools exist that support specific design tasks: however wider exploration of computer supported sketching is being hampered by the effort required to build the sketching software. Here we present a sketch tool framework, its implementation and evaluation. The implementation, InkKit, provides context free design spaces and a powerful, trainable and extensible modeless writing/drawing recognition engine. It reduces the development effort for a specific diagram type from thousands of lines of code to a few hundred. We evaluated our toolkit by asking fourth year computer science students to use InkKit to develop a diagram specific recognizer.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation (e.g., HCI)**]: User Interfaces - *graphical user interfaces.*

General Terms

Design, Human Factors.

Keywords

Sketch tools, hand-drawn diagrams, sketch recognition, pen computing.

1. INTRODUCTION

Designers consistently hand-draw sketchy preliminary diagrams before preparing formal computerized designs. The benefits of computer-based sketch tools have been demonstrated by a number of studies [3, 26, 30]. The software tools developed for these, and other, studies inform us about how designers can be supported by sketch software. However, most tools have focused on a particular domain and it is difficult to predict the transferability of outcomes across domains and when mixing techniques. For example, is the effect of computer beautification (tidying) of sketched user interface designs and directed graphs the same?

Developing a sketch tool is a non-trivial task: Freeform [30] is 12,000 lines of code. A toolkit approach should reduce the effort required and provides a consistent platform to conduct within-domain and across-domain inquiry. However Lank [20] reported that his framework required 2,000 lines of code and the diagram specific recognizers were between 2,000 and 5,500 lines each. Our framework and toolkit are more comprehensive, and subsequently reduce the code for diagram specific recognizers to a few hundred lines.

This work reports both a framework and toolkit for sketching. InkKit, the toolkit is a fully featured, extensible sketch environment that minimizes the development effort required to support a particular type of diagram. InkKit includes a novel two-view user interface [27] and a recognition engine that refines existing algorithms and adds new techniques for dividing drawing and writing ink, and recognizing complex components [11]. The acid test for a software toolkit is the reduction of development effort for others. Here we present an evaluation of InkKit where 4th year students successfully wrote diagram recognizers in just a few hours.

2. MOTIVATION

Research suggests that designers prefer to explore their design ideas in an informal environment [26, 37] and produce better results when their initial designs are hand-drawn [12, 28]. These studies have been conducted on quite concrete designs such as user interfaces (UIs). There are open questions about whether the same is true with more abstract diagrams such as UML diagrams and electrical circuit diagrams.

In addition, there are other sketching related issues yet to be explored. For example, many people are reluctant to share their scribbles with a superior or client – although they will get better feedback from a sketch [37]. Partial electronic tidying (beautifying) of a sketch may make the designer feel better about it; however the effect on the feedback loop is unknown. Likewise, as educational benefits ensue from automating (animating or executing) visual representations of algorithms, are there benefits to be had from automating sketches? Wong [37] suggested that there were benefits for interface design and a number of sketch tools have supported this type of interaction: but we are not aware of any formal studies on its effect. By reducing the effort required to develop sketch tools these and other issues can more easily be explored.

3. INTRODUCTORY EXAMPLE

InkKit is a fully-featured sketch toolkit based on the framework presented later in this paper. The fundamental goal is to minimize the effort required to support sketching a specific type of diagram: for example user interface designs. In this section we describe the steps required to create an InkKit plug-in library that recognizes and transforms the multi-page sketches shown in Figure 1 into the UIs shown below the sketches.

© Beryl Plimmer, Isaac Freeman, 2007
Published by the British Computer Society
People and Computers XXI – HCI… but not as we know it: Proceedings of HCI 2007
Linden J. Ball, M. Angela Sasse, Corina Sas, Thomas C. Ormerod, Alan Dix, Peter Bagnall, and Tom McEwan (Editors)

The UI plug-in library consists of: one interpreter plug-in that tailors the recognition engine for user interface diagrams; a set of example hand-drawn components (Figure 2): and two output plug-ins (one each for HTML and Java). UI interpreter (class) is 320 lines of code; half is declarations of domain information and components. For the most part this is simple 'copy and paste' code that could be generated by a wizard. The remainder interprets the output from InkKit's imbedded recognizer (details in Section 6.2): most components require only a few lines of code to pass the InkKit recognized component to the data structure for the output plug-ins. However this step allows the programmer to add code to deal with diagram specific functionality; for UI designs the interpreter generates data descriptions for off-page links from buttons and drop down lists (Figure 1).

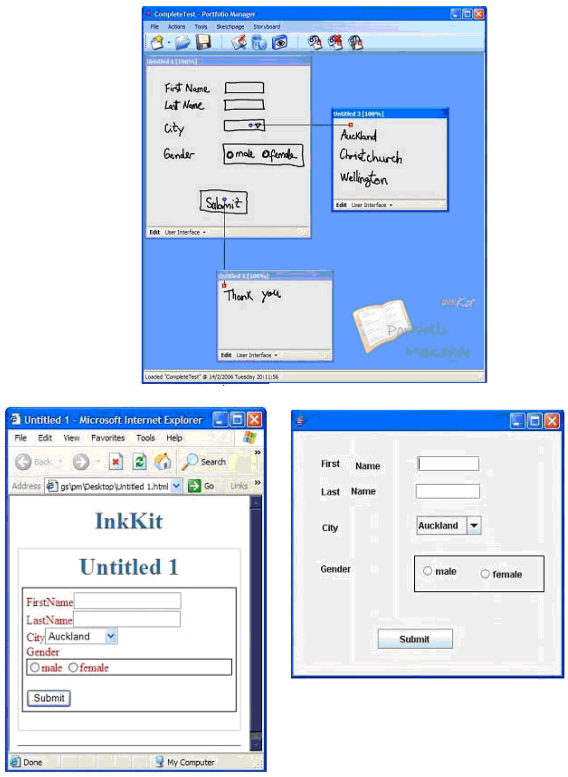


Figure 1. InkKit UI diagrams translated to HTML and Java.

A typical sketch recognition engine is thousands of lines of complex code. The 320 lines of simple code in the interpreter is all that is required to recognize 10 common UI components, inter-sketch links, and generate a data structure for consumption by the output generators or other plug-ins. A small number (2-5) of hand-drawn examples (Figure 2) of each component are also required. We stress that no complex recognition rules need to be written.

We have implemented two output plug-ins for the UI domain, HTML and Java forms. Each takes the same data structure from the interpreter; they are each approximately 500 lines of code. The plug-in programmer is presented with a collection of 'interpreted page' objects. It is straightforward to parse this collection and from it generate the UI, mainly generating the different component types and converting size and position data. Other attributes may be transferred (such as containment) depending on the particular requirements of the language. The HTML is immediately executable, including inter-page links, while the Java code is ready to compile. We have also implemented plug-ins for organization charts, undirected and directed graphs as examples of diagrams that have connected components based on spatial position, edges and directed arcs respectively [11]. Students created five further sets of plug-ins (see Section 7).

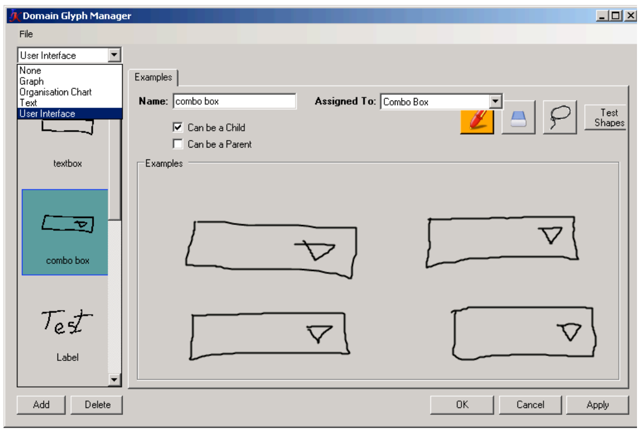


Figure 2. InkKit domain library UI showing example shapes.

4. REVIEW OF EXISTING TOOLS

To define a framework for sketch tools we examined a range of sketch tools to identify the generic functionality required. The framework focuses on the designer's interaction requirements and generalizing sketch recognition (the most difficult technical challenge for sketch tools). We used this framework to guide the development of InkKit.

Most of the sketch tools that have been developed support diagrams associated with Computer Science. Many are for UI design, for example Silk [19], Denim [22], Freeform [29], and Demais [3] or for UML diagramming Knight [7], Sumlow [5]. Architecture has also attracted interest [9, 35]. Satin[15] is an earlier sketch toolkit; the user interface is a single large drawing canvas and it includes a range of useful routines for the manipulation and recognition of drawing ink. It has been used successfully by Landay's group to explore sketching support. However, it does not include character recognition and is a programmer's library rather than an extensible environment. A number of other sketch toolkits or frameworks have been proposed. SketchREAD [1] is an innovative approach using Bayesian networks to recognize sketches; however it does not appear to support text on the diagrams. It requires the user to describe the diagram components with complex rules, whereas InkKit builds the rules from user examples. Likewise SketchiXML [6] defines a general purpose description language for sketches to support cross-platform implementations but neglects the recognition of text. Our work is most similar to Lank's [20] retargetable framework. While Lank's framework, like InkKit, recognizes text and drawing components, the amount and complexity of code needed to implement a domain extension is very high, leading Lank to question the viability of a framework approach.

From the literature on these tools we identify the functionality supported (Table 1). The common feature of all of these tools, not surprisingly, is a drawing space. There are two approaches to this: multiple pages and a storyboard, or one large space where navigation is aided by zooming or a radar window. Other common features are recognition to enable conversion from sketch to a formal design environment and automation.

Table 1. A summary of current sketch tools.

Tool	Domain	Sketch pages	Storyboard	Recognition	Animation	Export
Animated Figures [8]	Animation	X	X	Isomorphic mapping	X	
Damask [21]	UI Patterns	X				X
Demais [3]	Multi-Media UI	X	X		X	
Denim [22]	UI	X	zoom	Rubine's [32]	X	
The Electronic Cocktail Napkin [9]	Architecture	X	X			
FreeForm [29]	UI	X	X	Rubine's [32]+ character	X	X
Freeform UIs [16]	UI					
Knight [7]	UML Diagrams	X	radar	Rubine's[32]		X
MathPad^2 [17]	Maths	X	X	Draw & write	X	
Monet [38]	UI	X	X		X	
Motion Doodles [34]	Animation	X				
Silk [19]	UI	X	X	Rubine's [32]	X	
Sim-u-sketch [18]	Sim-u-link	X		Shapes and digits		
SketchiiXML[6]	UI Design	X		Cali [10]		
STCtools [25]	Device Design	X	X			
Sumlow [5]	UML	X		Apte [2] & Rubine's [32]		
Tahuti [13]	UML	X		Multi layer framework		
Retargetable Framework [20]	UML, maths formula, molecular diagrams	X		Multi step, includes character		

Sketch recognition is technically challenging. Many of the tools have used Rubine's [32] algorithm. This is a single stroke pattern matching algorithm that is simple to implement and train from examples. A range of other recognition techniques have been explored: for example, Ladder [14] proposes a general method for diagram recognition and Cali [10] has two versions, a non-trainable fuzzy-logic recognizer and a trainable Naïve Bayes algorithm. Shilman and Viola [33], suggest techniques for grouping ink. Few of these tools recognize hand-writing and drawing. Freeform [29] implemented modal drawing/writing interaction. Lank's [20] tool incorporates publicly available character recognition, while Math Pad^2 [17] includes limited drawing and writing recognition. Others have either ignored word input or support keyboard text entry. We contend that most non-trivial diagrams require words as an essential part of the description and it is distracting for the user to move between pen and keyboard. This suggests that a sketch recognition engine must be capable of recognizing diagrams containing both words and shapes.

There have been a number of studies on the efficacy of computer support for sketching [3, 30, 36] for UI design. These studies suggest that computer-based sketch tools do not disrupt the design process in the same way as standard computer design environments have been shown to [12]. However there is little evidence from other domains. Likewise automation (animation and execution) of sketched designs offers exciting prospects. A number of the current tools have explored this: for example, sketched UIs where the controls behave appropriately [19, 21] and animation of sketches [8, 31, 38].

Sketch tools present unique usability issues. Mankoff et al. [23] have looked extensively at user support for recognition correction – an important area as it will be some time (if ever) before consistently correct sketch recognition is achievable. We have recently reported on our usability testing of InkKit [27].

5. FRAMEWORK

We can discern from Table 1 that there is a core set of functional requirements for sketch tools. The key components are the UI and recognition engine. Support for normal functionality such as data storage and retrieval is assumed.

The user interface: two approaches to a user interface are apparent in the literature, a single large view or a two-view interface. A virtual page can be very large, yet display space is finite so large pages require support such as zooming or navigation aids like radar windows.

Two-view interfaces typically provide a place where a collection of sketch pages can be displayed and associations established between the pages. Many of the existing sketch tools refer to this as a storyboard, a term used by graphic designers. It suggests a linear arrangement of the sketches. Alternative arrangements could be a hierarchy or network where the relationships between pages may depend on their relative positions or explicit connectors. The semantics of page position and connectors is domain dependent, therefore not a part of the core functionality.

Regardless of size, sketch pages provide a place where users can draw and write with a pen much as they would on paper, with support for usual computer editing such as cut, copy, paste and undo. It is a moot question as to whether functional editing gestures (e.g., scribble over to delete) should be supported. Our experience is that when functional gestures work they are excellent, but when they fail users get very frustrated. Standard paper backgrounds such as grids and lines have been implemented by some tools and are likely to be useful. We envisage a range of standard backgrounds and the ability for users to create their own custom backgrounds. The position of background elements may be of interest to the recognition engine for particular diagram types.

The recognition engine: in a computer-supported environment a recognized sketch is more useful than an unrecognized sketch. Recognition is essential for more sophisticated support, it facilitates automatic conversion of sketches to formal diagrams and the automation of sketches, something that is not possible with paper equivalents [8, 19, 21, 31, 38].

A natural sketching environment consists of both words and characters; the recognition engine must manage this. To identify a structure for sketch recognition, consider the sketched diagrams in this paper, each of which has been constructed from individual ink strokes. Each stroke contributed to either a symbol or word. The meaning of the stroke is dependent on its shape and relationship with other strokes both spatially and within the context of the particular domain.

We can identify common properties of the ink. First, there is a distinct semantic difference between letter strokes that contribute to words and drawing strokes that contribute to symbols. Second, basic shapes, such as rectangles, are often meaningful on their own; however, frequently their meaning is derived from a particular spatial arrangement with other shapes either to form a more complex symbol or a relationship with another symbol. For example, a rectangle as a part of a standard UI, by itself represents an edit box, but with a triangle inside, it represents a dropdown. Third, connectors are common in diagrams: they represent relationships between elements.

Although there is a clear divide between the meaning of letter strokes and drawing strokes, we are accustomed to interleaving drawing and writing on paper and must be able to do the same with a computer sketch tool. To achieve smooth user-interaction, yet successfully recognize diagrams, we propose the following architecture for a recognition engine.

- First, classify the strokes as either writing or drawing strokes.
- Second, identify basic shapes such as lines, rectangles, and circles. Group the letters into words and recognize the words.
- Third, identify meaningful diagram components.
- Last, identify the relationships between components.

The first two recognition steps are independent of domain. The third and fourth steps are dependent on diagram type. However, diagram syntactic and semantic rules mean that the elements of a component have a discoverable relationship with sibling elements and the component. Consider the representations of a scroll bar in Figure 3.

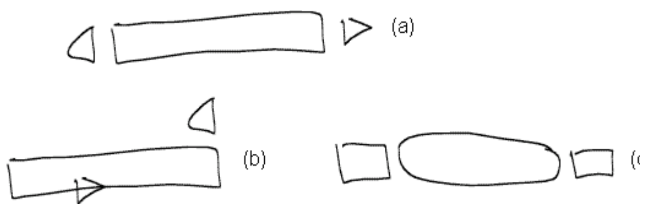


Figure 3. Component visualization.

If (a) is the defined representation, then (b) contains the same elements in a different spatial arrangement, whereas (c) has different elements but in the same spatial arrangement. To humans (c) is a closer match: unless (c) was itself a component they would classify it as a component of type (a) whereas they would be unlikely to classify (b) as type (a). Using these ideas, and identifying appropriate features, example-based component recognition can be achieved.

Other functionality of sketch tools is dependent on domain. Domain extensibility should include the ability to define: sketch page backgrounds, additional semantic rules to aid recognition, semantic rules defining relationships between sketches, the ability to generate data from a recognized sketch in a suitable format for other computer based tools such as diagram editors or programming IDEs, extensibility for interaction with the sketch for automation

6. OUR APPROACH

We have used this framework to built InkKit. To optimize adaptability and extensibility we have employed a component based architecture and implemented plug-in functionality. It has been developed for the Microsoft Tablet OS in Visual Studio .Net using C#. This has the advantages of providing a good hardware platform and the Ink SDK for basic ink data support and character recognition, but it restricts InkKit's use to the chosen operating system and hardware.

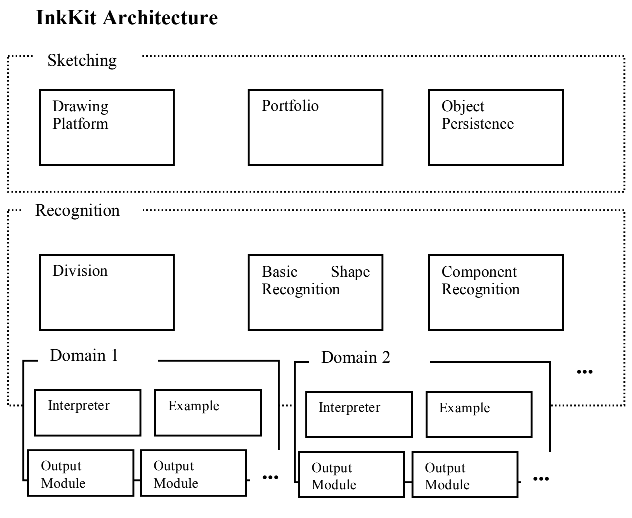


Figure 4. InkKit architecture.

Figure 4 is an abstract representation of the InkKit architecture. Below we briefly describe the major components of this: the user interface, recognition engine and extensibility for domain specific functionality.

6.1 User Interface

InkKit contains two main user interfaces: sketch pages and a portfolio manager. To maximize viewable space InkKit can be used with an auxiliary monitor (Figure 5). In this mode the portfolio manager resides on any standard output monitor and the sketch pages reside on the tablet (which accepts stylus input).

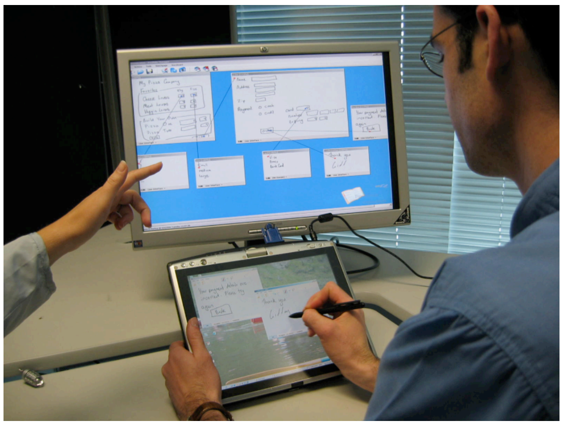


Figure 5. InkKit user interface.

To engender the feeling of working on paper the sketch pages are deliberately minimalist in appearance. Users can ink, erase, undo, redo and, edit by first selecting ink, and then resize, cut, copy and paste. Page size can be reduced or enlarged by dragging the sides or corners of the window. If reduction hides ink the virtual page is the minimum size to accommodate the ink and scroll bars are added to the page. Pages can be zoomed from a drop-down list.

The metaphor for the portfolio manager is that of spreading pages around a desk. Pages can be moved around the space and resized: resizing automatically zooms the page content to the available space. Links can be created between the pages

The user interface has been thoroughly usability tested (see [27]). The behaviour of the two visualizations (sketch pages and portfolio manager) was carefully tested. This resulted in sketch pages that are editable only in sketch-page view (not portfolio view) and the resizing of sketch pages acting differently in each view (resizing the sketch page, but zooming the sketch in portfolio view). Also we verified that the terminology used on the interface is understood across domains.

6.2 Recognition Engine

Our goal with the recognition engine is to be able to recognize forms and abstract diagrams containing shapes and words from user examples. From the architecture described above we developed a modular approach to the recognition (Figure 6). Much of the recognition is an implementation of well known techniques; however, the divider and component recognition are novel approaches to these problems.

Divider: The Tablet OS includes both a text recognizer and stroke divider. The text recognizer produces good results. We tested the divider on a variety of typical diagrams; 68% of drawing strokes were classified as text while 6% of letter strokes were classified as drawing. Given the nature of diagrams this error rate is unacceptable.

Analysis of sample text and drawing strokes identified features that we could use in conjunction with probabilities returned by the OS divider. Our divider first analyzes each stroke, assigning it a probability of being a letter, and then combines the probabilities of horizontally adjacent strokes. We reject text that is less than two letters in length so that simple shapes like circles are recognized (this can be overridden by the domain plug-ins). Using this approach we were able to achieve a much lower rate of false text (4%), while maintaining a low rate of false drawing strokes (10%). Strokes identified as text are passed to the OS for recognition.

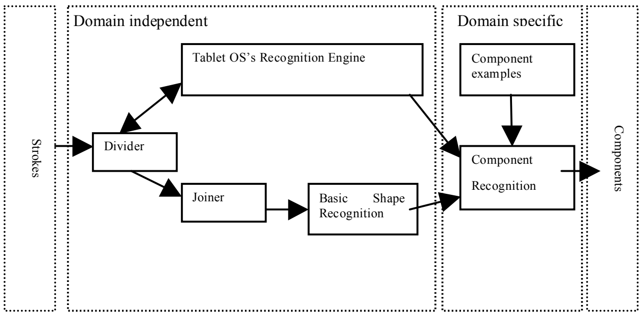


Figure 6. Recognition engine architecture.

Shape Recognition: The remaining strokes are drawing ink. Most diagrams consist of a small set of basic shapes that are simple outlines such as rectangles, triangles, circles and the like. Most of these can be drawn with one continuous pen stroke. A number of sketch tools (see Table 1) have used this characteristic and implemented Rubine's [32] algorithm for stroke recognition. Its attractiveness is the simplicity of implementation and that it is example driven.

After implementing Rubine's algorithm we identified three limitations to its use. First, it only caters for shapes created in one continuous stroke. Second, some of the features are tied to the absolute size of the shape. Third, the recognition, while good, was not accurate enough for our purposes.

Joiner: It is not always natural to draw a basic shape with a single stroke. Two approaches can be taken to this problem. Complex lines can be broken into component lines [4], or else component lines can be joined to form a single line. For example, a square could be divided into four straight lines, or the separate lines could be joined to form a square. We chose the latter approach joining adjacent successive strokes that do not already form a closed shape (such as a rectangle or triangle). Before joining, the recognizer's copy of strokes are rotated and trimmed to ensure a continuous smooth flow of points, but the user's view is left unaltered.

Basic Shapes: Solutions for the latter two problems (recognition dependent on size and higher accuracy) were produced by modifying the features used to classify shapes. We removed classification features that involved absolute size values, and inserted ratios in their place. As an example, width and height features were replaced by the ratio between the width and height. We also included features identified by other research [10] in relation to the convex hull and smallest enclosing rectangle. Ratios between the perimeters and areas of these shapes are useful because they ignore orientation issues. As an example, regardless of orientation, the ratio between the areas of the convex hull and enclosing rectangle is 1 for a square, 0.78 for a circle and 0.5 for a triangle.

Informal comparison testing between Rubine's algorithm and our enhanced algorithm on the same training and test sets from four users increased recognition rates from 44% to 92%. In addition, our approach offers more flexibility as users can construct their basic shapes with multiple strokes.

Components: At this point all of the strokes in a diagram have been identified as either text or a basic shape. The next stage is to recognize the diagram components; that is meaningful parts of the diagram that consist of one or more elements (shapes and words). Now information on the rules for the diagram domain is needed. User examples are used to extract these rules (Note: No domain specific code need be written for component recognition).

Each domain library is a collection of example components (Figure 2); the goal is to classify each part of the diagram as a component from the library. The component classification algorithm begins by constructing a graph of spatially related elements. From this graph element combinations are identified and evaluated against the domain library to produce the probability of the combination belonging to a particular component class.

This probability is computed using three sets of features. First, we calculate the probability of sibling relationships between each pair of elements existing in the component class examples. Features between the elements (enclosing, enclosed, near or intersecting) and their relative positions and orientation (centres, heights, widths and centre in relation to height) are used for this calculation. These probabilities are carried through to the next step.

Second, we compute the probability of each element existing as a part of each possible component type. This is done by combining the already computed probability of that element's sibling relationships, the shape of the element and the element's spatial position within the combination of elements.

Third, we compute probability tables of complete combinations in a similar manner to that used to classify basic shapes. Features used include properties of the bounding box of the combination and the density of shapes within it. Probabilities from these three sets of features are combined to produce a resultant probability of each element combination belonging to a specific component class.

Finally the graph is recursively searched for the most probable component matches, progressively assigning elements to components. As an element or set of elements is assigned to a component they are removed from the graph. Recognition of UI diagrams in our informal evaluation achieved a 95% success rate. We comment on the success of the recognition of other types of diagrams in the evaluation and discussion sections.

6.3 Extensibility

Extensibility is achieved through plug-in libraries that hold domain specific information: the example in Section 3 is such a library. A library has two or more parts, an interpreter that can extend the sketch recognition, output plug-ins that generate export data in appropriate formats and automation or other extensions. A domain can be added to InkKit by placing a library DLL into the InkKit program directory and then adding sketch examples of the components to the domain via the InkKit library interface (Figure 2).

The interpreter consists of basic information about the domain (such as whether the diagrams contain words and connectors) and a list of component names and descriptions. Through the interpreter, recognition can be enhanced by adding domain specific rules.

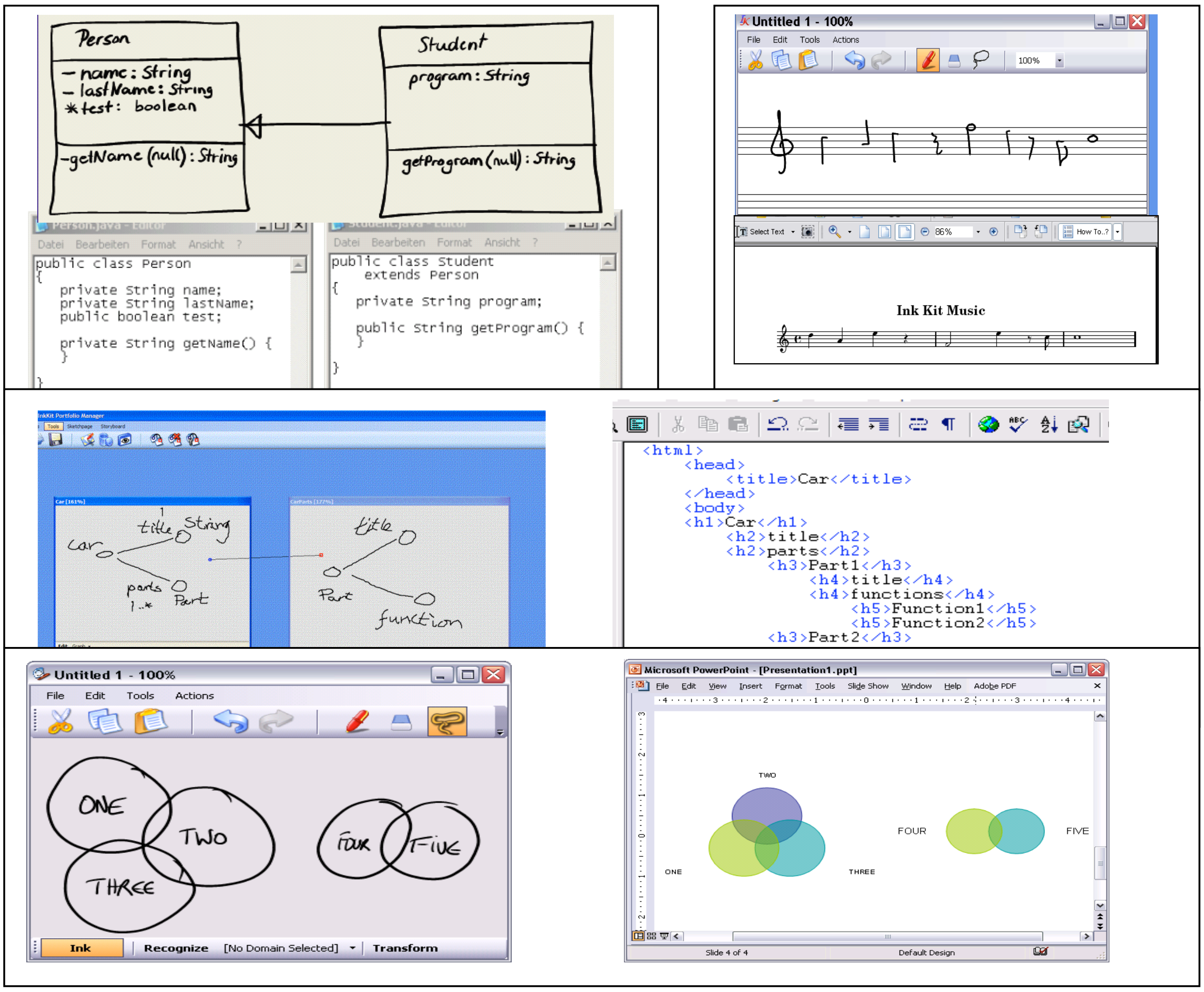
The output modules produce data in specific formats for other tools using the components identified by the interpreter. Each component has attributes that include recognition results, the raw ink, positional information, and connections with other components on the same page or other pages. The output module can apply additional rules depending on the output data requirements. Other extensions, for example for automation can also be added to the domain library.

7. EVALUATION STUDY

To evaluate InkKit, five 4th year computer science students each implemented a plug-in library as a part of an advanced HCI course. We suggested that 20 hours of work was the maximum that they should dedicate to the project. Each selected a different type of diagram and wrote both and interpreter and output class. We discussed with them the rationale behind InkKit and gave them a half hour overview on the software interface (API) for plug-ins. The resources they were given were: a compiled copied of InkKit (they had no access to the base code), three sample plug-ins (UI, hierarchy charts and graphs) and a brief written description of the API.

Table 2. Student libraries for InkKit.

Domain	Interpreter			To	Output		
	Code Lines	Number of Procedures	Maximum Complexity		Code Lines	Number of Procedures	Maximum Complexity
Venn Diagram	102	2	2	Powerpoint	293	3	5
UML Class Diagram	264	4	12	Java	830	5	16
UML Activity Diagram	281	3	11	Visio			
Hierarchical Visual Model	275	2	5	HTML	255	3	6
Music	459	4	23	Lilypond	510	4	21



```
public class Person
{
   private String name;
   private String lastName;
   public boolean test;

   private String getName() {
   }
}
```

```
public class Student
   extends Person
{
   private String program;

   public String getProgram() {
   }
}
```

```
<html>
   <head>
      <title>Car</title>
   </head>
   <body>
   <h1>Car</h1>
      <h2>title</h2>
      <h2>parts</h2>
         <h3>Part1</h3>
            <h4>title</h4>
            <h4>functions</h4>
               <h5>Function1</h5>
               <h5>Function2</h5>
         <h3>Part2</h3>
```

Figure 7. Student sketches and formal output.

They implemented a range of different types of diagrams (Table 2). One student chose music notation – he is also a music major (we added a simple music staff wallpaper to the sketch page for him). All students implemented a satisfactory interpreter and four implemented a satisfactory output module. The fifth was attempting to export to Microsoft Visio – he did manage to make drawing objects in Visio but could not arrange them correctly. Table 2 shows the lines of code and Cyclomatic Complexity [24] (<10 is simple, 11-20 more complex, 21-50 very complex, >50 untestable) required achieving the plug-ins. This is the code exactly as they submitted it without any optimization by us. The UML class diagrams and music are the most complex and these were created by the most conscientious students. Figure 7 is a selection of screen shots of the various diagrams taken directly from the students' presentations.

While all the students managed to recognize diagrams of their chosen domain, the accuracy varied from excellent for the Venn diagrams to nearly always needing manual correction for the Activity diagrams. The music scoring was limited to a small subset of the main music symbols that lay within the staff, not enough for any real music, however better than we had anticipated as musical notes are more closely related to writing than diagram components (on the example one of the notes is an octave out – a small 'bug' the student assured us).

From the students' plug-ins we identified that attaching a label (word) to an adjacent component was common across nearly all of the interpreters; we have since incorporated this functionality into the core recognizer. We also decided to examine more closely the syntactic and semantic rules around connectors as these are common to many diagrams. We have subsequently implemented identification of connectors and connection points into the recognizer and rewritten the graph interpreter reducing the lines of code required from 250 to 180 [11].

8. DISCUSSION AND FUTURE WORK

Hand sketching initial designs is a standard approach across a wide range of disciplines. The current range of computer-based sketch tools suggests that computer environments can emulate this by providing an ink-enabled interface for the sketch creation. Two methods of providing large spaces have been explored: a large space with navigation aids, or multiple spaces and a storyboard.

Our approach with InkKit is to provide a well-tested, easy to use interface that consists of two parts: variable sized, zoomable sketch pages; and a portfolio view where pages can be freely positioned and connections made between pages.

For computer-based sketch tools to be significantly more useful than pen and paper they must provide computational support that paper cannot. From Table 1 we can see that some of the areas under investigation are automatic conversion from sketch to formal diagram, and supporting automation; this requires a recognized sketch and knowledge of the underlying semantics.

Computer-based recognition of a sketch, particularly one that contains both drawing and writing elements, is difficult. Given the quite small set of basic elements from which most diagrams are constructed, and the spatial relationships between these elements, a general approach to recognition such as we have implement in InkKit is possible. Without regard for domain, ink is divided into writing or drawing strokes. Writing strokes are recognized by a text recognizer. Drawing strokes are joined to form basic shapes that are recognized by our enhanced version of Rubine's [32] algorithm, thus overcoming the main weakness of this algorithm.

Domain specific information is required to compose components from the words and basic shapes. In InkKit we do this from user examples, extracting from these examples the basic elements and their spatial relationships. Further, more specific, knowledge is required to convert a sketch to a formal representation or automate it. InkKit makes available the recognition information via an API so that these next stages are easy to program.

Robust recognition is essential and we are certain that further advances will be made. The component architecture of InkKit's recognition engine is such that alternative approaches can be evaluated against a variety of domains and as improvements are identified components can be replaced. However, InkKit is already more advanced in this respect than other tools in that it offers modeless writing and drawing and example driven recognition.

Students implemented a number of libraries to evaluate InkKit. Their success demonstrates the viability of a toolkit approach to sketching tools. InkKit has successfully handled a range of domain independent and domain specific issues.

Further work is needed to explore other sketch-tool related issues such as: eager versus lazy recognition, the timing and effect of beautification, and the effect of sketch automation. A toolkit approach means that robust comparative studies can be undertaken isolating the particular variables of interest.

9. CONCLUSIONS

The framework defined here has been demonstrated with the implementation of InkKit. InkKit provides the essential functionality required for sketching and has been successfully used to implement diagram recognizers in eight domains, and these diagrams have been converted into nine different formats. The skill and knowledge required to implement a plug-in is only that which we would expect from a graduate computer science student. In addition, the quantity of code and time is minimal. This general approach to sketch tools affords more robust evaluation and exploration of computer supported sketching.

This framework and InkKit open the way for more rapid exploration of computer-supported sketching and computational support of sketches such as animations, simulations and execution.

A copy of InkKit can be obtained by emailing the first author.

10. ACKNOWLEDGMENTS

We would like to acknowledge the contributions of Ronald Chung, Peter Mirica, Mark Young and Gene Tang who have worked on various previous versions of InkKit. Also the students who developed the plug-ins for InkKit for their enthusiasm and helpful feedback on the project.

11. REFERENCES

[1] Alvarado, C., and Davis. R.. SketchREAD: A multi-domain sketch recognition engine. In *Proceedings of the 17th annual ACM symposium on User interface software and technology (UIST 2004)*. ACM Press, New York, NY, 2004, 23-32.

[2] Apte, A., Vo, V., and Kimura, T. D. Recognizing multistroke geometric shapes: An experimental evaluation. In *Proceedings of the 6th annual ACM symposium on User interface software and technology (UIST 1993)*. ACM Press, New York, NY, 1993, 121-128.

[3] Bailey, B. P., and Konstan, J. A. Are informal tools better? Comparing DEMAIS, pencil and paper, and Authorware for early multimedia design. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (*CHI 2003)*. ACM Press, New York, NY, 2003, 313-320.

[4] Calhoun, C., Stahovich, T. F., Kurtoglu, T., and Kara, L. B. Recognizing multi-stroke symbols. In *Proceedings of the AAAI spring symposium on sketch understanding*. 2002.

[5] Chen, Q., Grundy, J., and Hosking, J. An E-whiteboard application to support early design-stage sketching of UML diagrams. In *Proceedings of the IEEE symposium on Human centric computer languages and environments*. IEEE Computer Society, Washington, DC, 2003, 219-226.

[6] Coyette, A., Faulkner, S., Kolp, M., Limbourg, Q., and Vanderdonkt, J. SketchiXML: Towards a multi-agent design tool for sketching user interfaces based on USIXML. In *Proceedings of the 3rd annual conference on Task models and diagrams (TAMODIA 2004)*. ACM Press, New York, NY, 2004, 75-82.

[7] Damm, C. H., Hansen, K. M., and Thomsen, M. Tool support for cooperative object-oriented design: Gesture based modelling on and electronic whiteboard. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (*CHI 2000)*. ACM Press, New York, NY, 2000, 518-525.

[8] Davis, J., Agrawala, M., Chuang, E., Popovic, Z., and Salesin, D. A sketching interface for articulated figure animation. In *Proceedings of the 2003 Eurographics/ SIGGRAPH symposium on Computer animation*. Eurographics Association, Aire-la-Ville, Switzerland, 2003, 320-328.

[9] Do, E. Y. L., and Gross, M. Thinking with diagrams in architectural design. *Artificial Intelligence Review, 15,* (2001), 135-149.

[10] Fonseca, M. J., Pimentel, C., and Jorge, J. A. CALI: An online scribble recognizer for calligraphic interfaces. In *Proceedings of the AAAI spring symposium on Sketch understanding*. 2002.

[11] Freeman, I., and Plimmer, B. *Connector semantics for sketched diagram recognition*. In *AUIC 2007*. (Ballarat, Australia, 2007). ACM Press, New York, NY, 2007.

[12] Goel, V. *Sketches of Thought*. MIT Press, Cambridge, MA, 1995.

[13] Hammond, T., and Davis., R. Tahuti: A geometrical sketch recognition system for UML class diagrams. In *Proceedings of the 2002 AAAI spring symposium on Sketch understanding*. 2002.

[14] Hammond, T., and Davis, R. LADDER: A language to describe drawing, display, and editing in sketch recognition. In *Proceedings of the international conference on Computer graphics and interactive techniques*. ACM Press, New York, NY, 2003, Article No. 27.

[15] Hong, J. I., and Landay, J. A. SATIN: A toolkit for informal ink-based applications. In *Proceedings of the 13th annual ACM symposium on User interface software and technology*. ACM Press, New York, NY, 2000, 63-72.

[16] Igarashi, T. Freeform user interfaces for graphical computing. In *Proceedings of the 3rd International Symposium on Smart Graphics*. Springer, Heidelberg. 2003.

[17] Joseph J., LaViola Jr., J. J., and Robert, C. Z. MathPad2: A system for the creation and exploration of mathematical sketches. *ACM Transactions on Graphics, 23,* 3 (2004), 432-440.

[18] Kara, L. B., and Stahovich, T. F. Sim-U-Sketch: A sketch-based interface for SimuLink. In *Proceedings of the working conference on Advanced visual interfaces*. ACM Press, New York, NY, 2004, 354-357.

[19] Landay, J., and Myers, B. Sketching storyboards to illustrate interface behaviors. In *CHI 1996: Conference companion of the SIGCHI conference on Human factors in computing systems.* ACM Press, New York, NY, 1996, 193-194.

[20] Lank, E. H. *A retargetable framework for interactive diagram recognition*. In *Proceedings of the seventh international conference on Document analysis and recognition (ICDAR 2003).* IEEE, 185-189.

[21] Lin, J., and Landay, J. A. Damask: A tool for early-stage design and prototyping of cross-device user interfaces. In *CHI 2003 workshop on HCI Patterns: Concepts and Tools*. (Fort Lauderdale, Florida, 2003).

[22] Lin, J., Newman, M. W., et al. Denim: Finding a tighter fit between tools and practice for web design. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI 2000).* ACM Press, New York, NY, 2000, 510-517.

[23] Mankoff, J., Hudson, S. E., and Abowd, G. D. Providing integrated toolkit-level support for ambiguity in recognition-based interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI 2000).* ACM Press, New York, NY, 2000, 368-375.

[24] McCabe, T. J., and Watson, A. H. Software complexity. *Crosstalk, Journal of Defense Software Engineering, 7,* 12 (1994), 5-9.

[25] Nam, T. -J. Sketch-based rapid prototyping platform for hardware-software integrated interactive products. In *CHI 2005: Conference companion of the SIGCHI conference on Human factors in computing systems.* ACM Press, New York, NY, 2005, 1689-1692.

[26] Newman, M. W., Lin, J., et al. DENIM: An informal web site design tool inspired by observations of practice. *Human-Computer Interaction, 18,* 3 (2003), 259-324.

[27] Plimmer, B., Tang, G., and Young, M. Sketch tool usability: Allowing the user to disengage. In *People and Computers XX (Proceedings of HCI 2006).* Springer, London, 2006.

[28] Plimmer, B. E., and Apperley, M. Evaluating a sketch environment for novice programmers. In *CHI 2003: Extended abstracts of the SIGCHI conference on Human factors in computing systems.* ACM Press, New York, NY, 2003, 1018-1019.

[29] Plimmer, B. E., and Apperley M. Software for students to sketch interface designs. In *Proceedings of the IFIP conference on Human-computer interaction (INTERACT* 2003). IOS Press, 73-80.

[30] Plimmer, B. E., and Apperley M. INTERACTING with sketched interface designs: an evaluation study. In *CHI 2004: Extended abstracts of the SIGCHI conference on Human factors in computing systems.* ACM Press, New York, NY, 2004, 1337-1340.

[31] Rogers, W. J. Living Ink: Implementation of a prototype sketching language for real time authoring of animated line drawings. In *Proceedings of the Eurographics workshop on Sketch-based interfaces and modeling.* Eurographics Association, 2006.

[32] Rubine, D. Specifying gestures by example. *ACM SIGGRAPH Computer Graphics, 25*, 4 (1991), 329-337.

[33] Shilman, M., and Viola, P. Spatial recognition and grouping of text and graphics. In *Proceedings of the Eurographics workshop on Sketch-based interfaces and modeling.* Eurographics Association, 2004.

[34] Thorne, M., Burke, D., and Panne, M. v. d. Motion doodles: An interface for sketching character motion. *ACM Transaction on Graphics, 23,* 3 (2004), 424-431.

[35] Trinder, M. The computer's role in sketch design: A transparent sketching medium. In *Computers and building: CAAD futures 99*. Kluwer, Atlanta, 1999.

[36] Walker, M., Takayama, L., and Landay, J. A. High-fidelity or low-fidelity, paper or computer medium? In Proceedings of the *Human factors and ergonomics society 46th annual conference*. (Baltimore, 2000).

[37] Wong, Y. Y. Rough and ready prototypes: Lessons from graphic design. In *CHI 1992: Posters and short talks at the SIGCHI conference on Human factors in computing systems.* ACM Press, New York, NY, 1992, 83-84.

[38] Yang, L., and James, A. L. Informal prototyping of continuous graphical interactions by demonstration. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*. ACM Press, New York, NY, 2005, 221-230.