Cross-Domain Diagram Sketch Recognition

Paul Schmieder[1], Beryl Plimmer[1], Jean Vanderdonckt[2]

[1]University of Auckland,
Private Bag 92019
Auckland, New Zealand
{psch068@ec, beryl@cs}.auckland.ac.nz

[2]Université catholique de Louvain
Place des Doyens 1
1348 Louvain-la-Neuve, Belgium
jean.vanderdonckt@uclouvain.be

Abstract

Diagrams are often used to model complex systems: in many cases several different types of diagrams are used to model different aspects of the system. These diagrams, perhaps from multiple stakeholders of different specialties, must be combined to achieve a full abstract representation of the system. Many CAD tools offer multi-diagram integration, however sketch-based diagramming tools are yet to tackle this difficult integration problem. We extend the diagram sketching tool InkKit to combine software engineering sketches of different types so as to automatically generate one or multiple code-specific outputs which interact with each other. Our extensions support software design processes by providing a sketch-based approach that allows the creation of multiple outputs interacting with one another from the inter-linked diagram input.

1. Introduction

The use of pen and paper is the most natural way to draft ideas in a non-digital environment and the methods to accomplish the same tasks on the computer world should be similar. This interaction can be achieved by using a digital stylus rather than keyboard and mouse. Studies show that, while there is still no equality in familiarity and intuitiveness between the classical way to bring ideas down and its digital counterpart [2, 24], there is a clear preference for computer-based sketch tools over their widget based equivalents because of the more intuitive interaction offered by the digital pen [8, 21].

Computer based sketch tools offer different features depending on the program's domain and implemented functionality. While simple implementations of sketch tools offer a canvas to draw on, more sophisticated ones additionally recognize the sketches. Due to the diversity of possible sketches the demands on the underlying algorithms are high. On one hand they have to cover all possible shapes and on the other hand they have to successfully differentiate between the shapes, even when they look very similar. Once the sketch is recognized, it can be interpreted and converted into symbolic expressions which represent the user's intent.

There are different diagram domains which can be sketched in a digital environment such as user interface (UI) and entity-relationship (ER) diagrams. The former outlines the design of a graphical user interface; the latter are used to specify database drafts at a preliminary stage, including the relationships between their components. While a number of sketch tools can recognize a specific type of diagram and translate it into a formal representation or, in the case of software, generate code, we are not aware of any sketch tools that combine different types of diagrams to generate a more compete model or system.

InkKit [18] is a software toolkit used to recognize and convert user-drawn sketches into other representations. Its layered design allows an easy and intuitive implementation of new domains. Each domain consists of one recognizer and multiple output modules. This means that once a domain-specific interpreter is implemented, output can be generated in various formats. For example, a user interface sketch can be converted into HTML, Java or other "output specific" code after being recognized. InkKit can recognize sketched diagrams which are split into several parts as shown in Figure 1.

In this project we extended InkKit so that it combines the interpretation results of different types of diagrams. These independent diagrams can interact with each other because the necessary information is exchanged during the interpretation process. Our exemplar is ER and UI diagrams that are recognized and used to generate a database and connected user interface. Afterwards the user can enter data in the UI which is then transmitted to the database. This way of combining recognition results from different hand-drawn sketches enables new opportunities for collaborative work. At a preliminary stage of design people with skills from different areas could work together or independently. Once they are finished, they could import their sketched ideas into one project and link the related parts which can then be further processed.

VL/HCC Workshop: Sketch Tools for Diagramming
Herrsching am Ammersee, Germany
15 September 2008
Editors: Beryl Plimmer & Tracy Hammond

In the next section the related work is presented, followed by the detailed description of our approach. In the fourth section the work is discussed before the final conclusions and future work.

2. Motivation

Complex systems, in a wide variety of different domains, including architecture and engineering, natural systems and software, are often defined by abstract models. Because of the complexity of such systems, different models are used to describe different aspects of the system. Yet the system itself is a complex interplay of these different models. In many cases diagrams are used as the visualization of the model.

Software systems are a particularly interesting example of abstract models and diagrams because the model can be used to generate the system. Increasingly software modeling tools support code generation. Yet, because of the formality and constraints of these tools such formal models are rarely used during initial design. Instead people revert to using whiteboards and scraps of paper. Sketch tools aim to bridge this gap and sketch toolkits with configurable recognition engines are allowing us to more easily explore the intersection between tools, models and systems.

There are various methodologies, models and diagrams used to describe software systems (for example UML). However, at the most basic level, 'ordinary' software systems consist of a user interface, data and processes. Figure 2 shows a simple set of three diagrams that could be used in a first interaction of a design. Our goal is to take a set of related diagrams like this and use them to generate the software system.

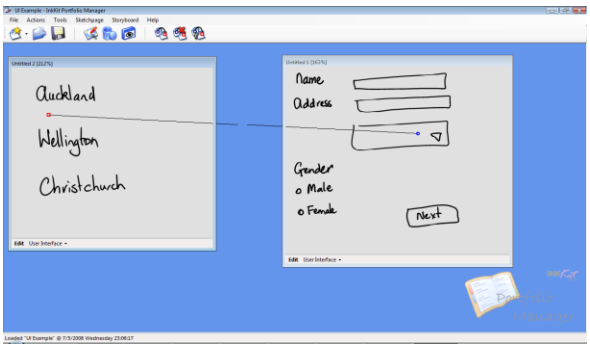

Figure 1. InkKit portfolio manager which contains two sketch pages connected via a rubberband

3. Related Work

Sketch tools can be differentiated by their basic features such as their recognition engine, their ability to process text or the domains they recognize. The latter can be furthermore subdivided into single- and multi-domain recognizers. The recognition engine is the component of the sketch tool which is responsible for the scope of domains. There are two different engine designs; domain specific and generic. While the latter are designed to recognize different diagram types, specific engines are restricted to one domain.

The first published sketch tools had recognition engines dedicated to one particular domain. For example, Silk [10], as one of the first, was published in 1996 and was specifically designed to recognize UI diagrams. Four years later Knight [6] followed, and another two years later Tahuti [9] was designed to recognize UML class diagrams. More recent sketch tools are DEMAIS [3], Freeform [20] and SketchiXML [5] which all recognize UI diagrams.

Examples for sketch tools which have a generic recognition engine are Lank's framework [12], SketchREAD [1] and InkKit [18]. Additional domains can be added to each of these tools. However, the implementation complexity varies significantly from tool to tool. Lank's framework is the most expensive one to extend in relation to code complexity and amount of code.

The majority of diagrams recognized by the sketch tools are from the fields of Computer Science and Engineering; user interface diagrams [2, 4, 11, 13, 22], UML class [6, 9] or circuit diagrams [1].

To our knowledge no sketch tool is capable of recognizing diagrams from different domains in one step and linking the generated output. However, when using InkKit, multiple diagrams from the same domain (for example linked UI pages Figure 1) can be recognized and a unified representation can be generated as if it were drawn in one sketch. Denim [16] achieves a similar result by providing a very large drawing space that is viewed at different levels of abstraction. Actions at the higher levels of abstraction determine the overall website and page attributes while the detailed levels translate to the page component. These automatically generated applications can interact with each other because the necessary information was exchanged during the interpretation process. In this project we extend InkKit to handle multiple sketches from two different domains and automatically generate output that reflects their cross-relationships.

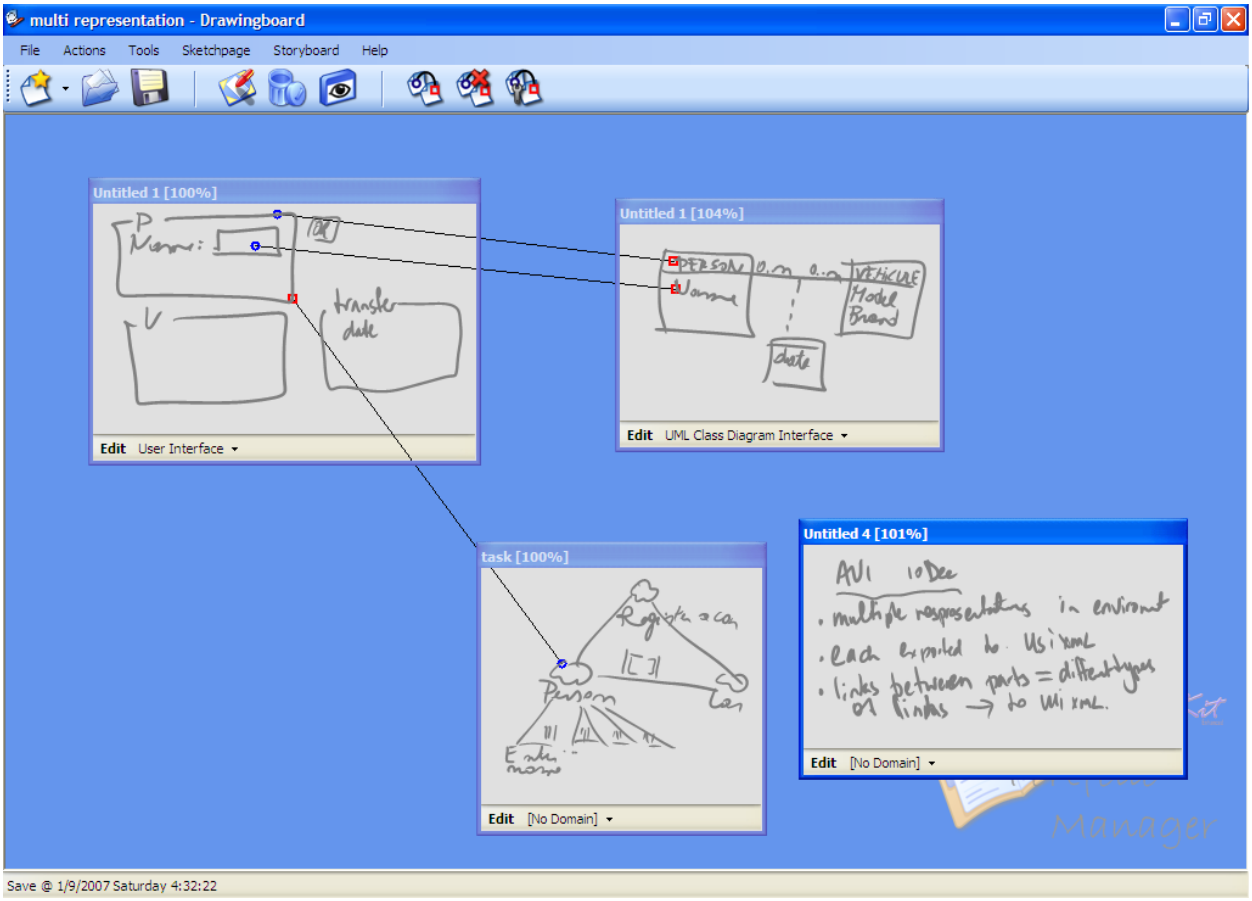

Figure 2. Sketches of related user interface, class diagram and process hierarchy and notes page

InkKit Overview

Two main user interfaces represent InkKit's graphical front end: sketch pages and a portfolio manager. The portfolio acts as a container for the sketch pages (see Figure 2). This design is robust and well tested [19] and enables intuitive user interaction. In addition to basic functions such as sketch page resizing, moving and zooming, connectors between the sketch pages can be added. They represent a relationship between the connected pages. For example, Figure 1 shows two sketch pages which are connected with each other. The left sketch page represents a list of city names which will be added to the "connected" combo box when both user interface diagrams are recognized and interpreted. The ability to merge sketches enables an easy, clear and well-arranged way to draw comprehensive diagrams. There is no beautification applied to the sketches within InkKit in order to preserve their hand-drawn appearance [2, 24]. Beautification occurs naturally when the recognizer output is rendered in another tool – and this may be enhanced by applying layout constraints as we have done by including ALM [14] as a part of the Java output.

Figure 3 shows InkKit's overall architecture. The recognition process consists of two main parts: the domain-independent and the domain-dependent. Starting with the independent part, the sketched strokes are classified either as text or shape strokes. This is done with the help of a decision tree which uses features such as time, sketching speed and spatial relationships for the classification [17]. Those strokes recognized as letters are grouped into single words and recognized by the operating system text recognition engine.

Sketched shapes can consist of more than one stroke. In order to use Rubine's [23] single stroke algorithm, the strokes which constitute one shape have

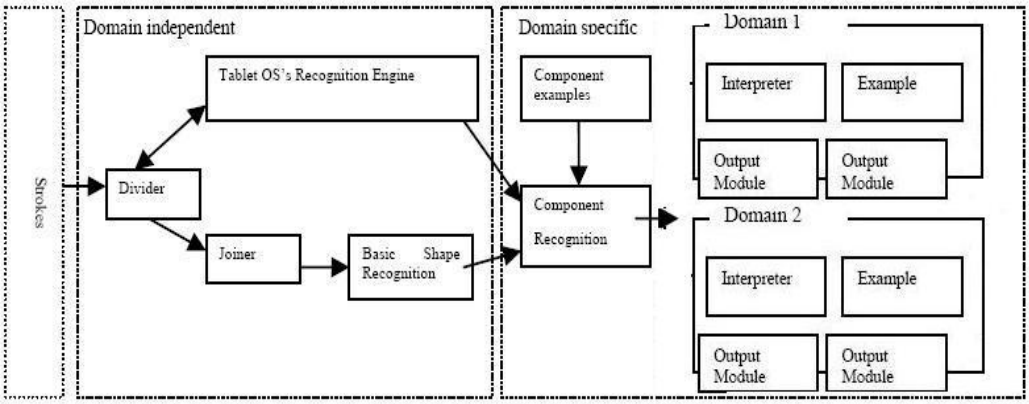

Figure 3. Architecture InkKit

to be joined. This is done by iterating through the strokes and measuring the distance between the endpoints of two consecutively drawn strokes. If the distance is within a predetermined threshold, both strokes are joined by replacing them by a single composed stroke [7].

After being joined the shapes are recognized. This domain-independent process recognizes basic shapes rather than complex domain-specific components. This decomposition of the sketched complex shapes is possible because all of them consist of a set of basic shapes.

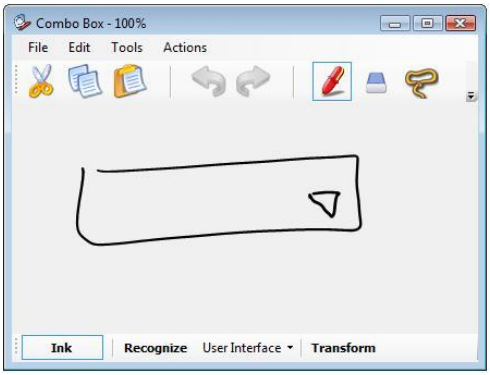

Figure 4. A combo box consisting of the basic shapes rectangle and triangle

For example, Figure 4 shows the user interface component "Combo Box". In the first recognition step this complex shape is decomposed into a triangle and a rectangle, which then get recognized independently. The basic shapes are taken from a predefined set such as circle, rectangle and line which can be extended by the user.

After all sketched strokes are recognized as basic shapes (except those recognized as text strokes), these results get handed over from the generic recognition engine to the domain-specific one.

Each domain consists of components which the user has predefined in the form of sketches. These components are stored in the specific domain library. InkKit's current version consists of nine domain libraries: activity diagram, directed graph, undirected graph, entity-relationship diagram, organization chart, parsimonious data model graph, UML class diagram, user interface and Venn diagram.

The first task of the domain-specific recognizer is to cluster the basic shapes into groups based on the basic shapes' spatial relationships. The computation of these relationships is derived from spatial features such as near or intersecting, relative position and orientation. The result of these likelihood computations is then used to calculate the probability of the basic shape groups to be part of a domain specific complex component.

Finally, using the likelihood calculation results of the basic shape groups, a hypothesis space is built which includes all these possible group combinations. Thereby groups are joined together to a complex shape based on several factors such as their spatial relations and their bounding box properties. Since a group of strokes can already be a complex shape, a combination can consist of one or more groups. The next step is to compute probability tables for each of these combinations of possible complex shapes. After all combinations are classified, the one with the highest probability gets assigned to its associated complex shape and is taken out of the hypothesis space. This association process is repeated in a descending order of the combination's probabilities until all sketched strokes are assigned.

In order to implement a new domain in InkKit, an interpreter describing the domain's properties and sketched examples of all of the components of that domain have to be added. In addition to the examples

of the domain components, they have to be defined in the interpreter. Furthermore, the relations of the components and the domain-specific data model have to be defined in the interpreter. The expense of implementing such an interpreter significantly depends on its scope of services. For example, the most compact one consists of 150 lines of code (InkKit's organization chart interpreter) and the most complex one of 880 (InkKit's ER interpreter).

Once the interpreter is implemented, output modules can be added to generate a representation of the sketches in a specific format. Again, the scope of services provided by the output module will determine its complexity and size. Existing output modules range from 130 lines of code (InkKit's graph text output module) to 350 (InkKit's ER Microsoft Office Access output module).

InkKit's general design is a composition of layered code segments which communicate through interfaces. Thereby a code segment is responsible for a specific task. This enables an easy modification of the single layers to integrate and test new technologies.

4. Cross Domain Requirements

To recognize relationships between diagrams of different types and intelligently generate output that leverages these relationships, the recognition engine must pass information between the different types of diagrams. There are two main approaches to enable the information exchange: specifying and implementing a communication protocol, or providing a shared object acting as an information carrier which is passed to every interpreter.

Communication protocol

The communication protocol is a more complex method than the shared object. It would enable a direct communication between the single interpreters. This follows an "information on demand" approach which means that an interpreter could ask for the needed information at any time.

The detail of the protocol is as follows; first the interpreter would start to analyze the results from the recognition. If the interpreter discovers that it needs additional information from another interpreter, it generates a message, puts it on a communication stack and places itself on hold until it gets the required information. Then the next interpreter would start and check the communication stack whether there is a message which is addressed to it. If so, the interpreter checks whether it could provide the necessary information at this stage. If it can offer a satisfactory answer it generates a message containing the information, puts it on another communication stack and deletes the message which it just answered. If the interpreter cannot offer the information it would ignore the request. This process would be repeated until every interpreter has run at least once and the communication stacks are empty.

Information carrier

An information carrier is a data structure which contains all the information about a sketch. It is created independently as a step of the page interpretation without reference to related pages. The use of an object as the information carrier instead of a communication protocol results in several disadvantages:

- The order of interpretation must be predefined by the user.
- Once the order is determined, the interpretation sequence is fixed.
- Each sketch is interpreted exactly once. If the needed information is not available at the time of interpretation, there is no possibility to go back when it is available.
- The interpreters have to store all information which could be needed by other interpreters which results in a waste of memory.
- The interpreter has to find the information in the object.

However an information carrier also has the advantages of being easier to implement and extend. This extensibility is important when, as in this case, the problem space is not well understood. It also is self-contained, not requiring information about the problem, the data, or how the data will be used.

5. Our Approach

In order to enable the interpretation of diagrams from different domains in a single step we need to extend InkKit and implement appropriate output modules. As an example we have chosen to take two of the three basic system models by combining the data representation with the user interface (we simulate the process information). The data is described in an ER diagram and the UI is a simple sketched representation (see Figure 5). InkKit's recognition engine has to be adjusted to support multiple types of diagram interpreters in the same portfolio and information exchange between the interpreters has to be established. The adjustments include the extension of

the domain interpreter, the addition of a loop to control the interpretation process, and the implementation of a MySQL output module. To enable the information exchange between the interpreters a new data structure must also be introduced to act as an information carrier. To store and retrieve information from this data structure the UI's Java output module and the ER's MySQL module require modifications. Finally, adjustments were made in both modules to interpret and use the exchanged information.

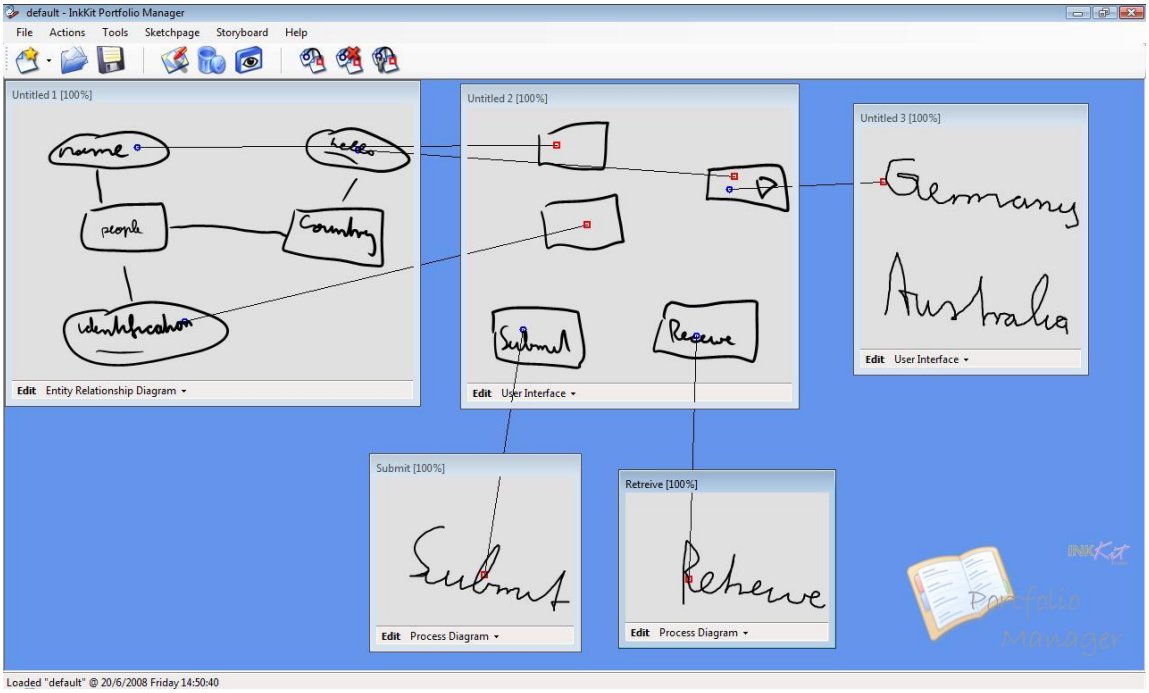

Figure 5. InkKit's portfolio manager including a sketched ER diagram, user interface and processes

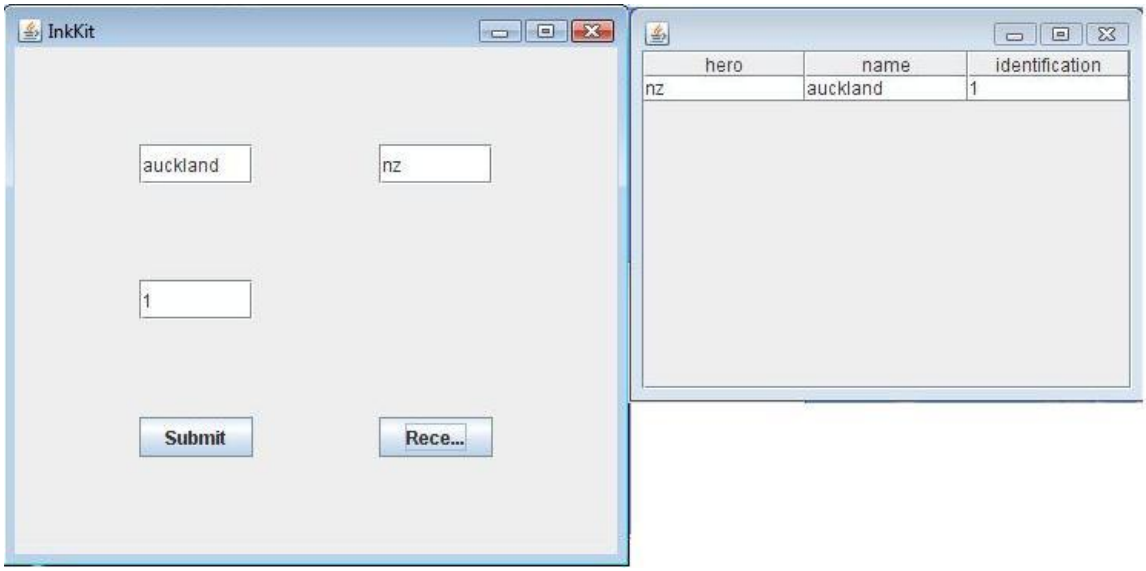

Figure 6. User Interface and Database table automatically generate from the sketches in Figure 5

Multi interpreter list

In order to interpret diagrams from different domains, the interpreters from the sketched diagram domains must be loaded. Previous implementations of InkKit were only able to interpret one domain at a time, so only one interpreter was loaded. For this project, the variable which stored the interpreter was replaced by a list. One instance of each interpreter is loaded and passed to the next program layer, the recognizer. After all sketches were recognized, they have to be interpreted. While the recognition process assigns the sketched components to their most likely predefined matches, the interpretation brings a meaning into the overall sketch. For example, after the ER diagram shown in Figure 7 is recognized, its components are known, i.e. the two entities, two attributes and 3 connectors. It is then the interpreter's task to give this composition of elements a meaning. In this example the interpreter would create a one-to-many relationship between the two entities "address" and "street", assign attribute "one" to "address" and "two" to "street" and determine that "one" and "two" are primary keys.

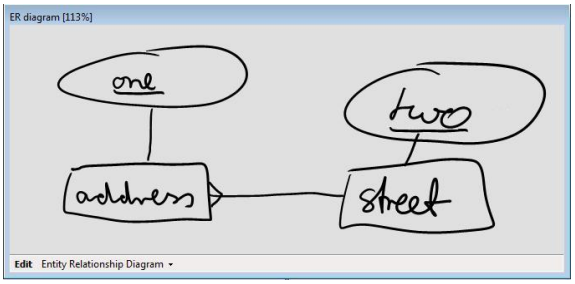

Figure 7. Entity Relationship diagram

Domain interpretation loop

The loop controlling the interpretation of the sketches had to be extended to handle more than one diagram domain. The loop's purpose is to process relations between different sketch pages which are indicated by connectors (called rubber bands) between the sketches (see Figure 2). Until that point in time, the loop could handle one interpreter and relate the corresponding sketches from the domain (see Figure 2).

The method which contains the loop calls itself recursively until all relations between the sketches are discovered. The loop extension includes the implementation of code which supervises the loop and controls the sketch order, meaning that diagrams from the same domain are processed consecutively.

After the sketches are interpreted, output modules can generate format-specific code based on the interpretation results. Every domain has output modules which generate specific code; for example, the UI domain has two output modules which generate Java code and HTML code.

We implemented an enhancement to the Java code output module to improve the aesthetics of the generated output. Until this point, the Java output module has not used a manager to organize the GUI's layout. There are several layout managers available such as Gridbag Layout Manager and ALM [14]. ALM is focused on the tabstops between cells rather than on the cells of grid like the Gridbag Layout Manager. This generalization of grid-based layouts makes ALM the more powerful manager in terms of adaptive layout resizing [15]. By using the layout manager the form appearance is enhanced, due to the standardized sizes of components of the same type and the harmonized positions of the components.

A MySQL output module was added as part of the ER domain as it provides an ease interface to the Java front end that we planned. The implementation consists of 650 lines of code, making this module one of the more complex InkKit output modules. The reason for its size is the complexity of ER diagrams - different sketch components are connected with each other and therefore form a single, complex structure rather than a collection of independent components.

InkKit extensions

After the necessary changes in InkKit's implementation were made, the communication between the interpreted sketches had to be established in order to exchange information. Thereby, several challenges had to be overcome which are explained in this section.

If diagrams from different domains are interpreted in one step, information can only be exchanged sequentially. This makes it necessary to interpret diagrams in a particular order.

Since InkKit's design follows a modular approach, it only calls the interpreters and hands over the needed information. This means that InkKit does not actively coordinate messages between the different interpreters, which is why the interpreters have to coordinate the communication by themselves. We considered two approaches to this, as describe above, a communications protocol or information carrier object passed between the interpreters. We decided to implement the information carrier object because of easier integration into InkKit's current architecture, the lower degree of implementation complexity and the lower complexity for maintenance.

InkKit has no information about the interpretation at any stage due to the modular design which

encapsulates the recognition from the interpretation. Therefore it cannot know the order in which the sketches have to be recognized. To give the user the possibility to order the sketches manually and not to rely on a random sequence, a GUI listing the used interpreters was implemented (see Figure 8). With its help the user can reorder the interpreters. In case there are several sketches from the same domain (i.e. they have the same interpreter) they are ordered in a consecutive order. Thereby the order from sketches belonging to the same domain cannot be influenced.

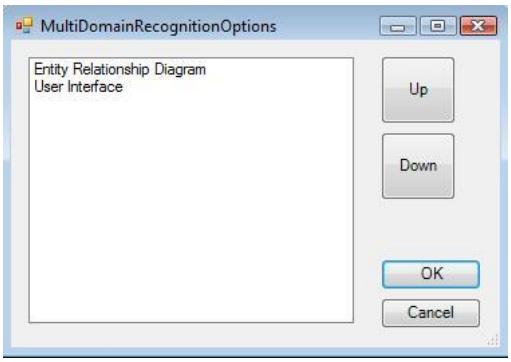

Figure 8. Interface to order the interpreters

Since it cannot be guaranteed that the user knows the correct order and that the information can be provided when needed, the interpreter was designed to be fault-tolerant. This means that if the information is not available, the interpreter produces an incomplete result which is then further used by an output module. Afterwards, the gaps in the generated output code can be manually completed by the user.

We took a pragmatic approach to the problem of what information to provide: since the impact on performance is low and enough memory will be available all information which could be relevant at a later time is stored.

Since InkKit does not know how much information will be stored, it must provide a dynamic structure which can handle as much information as necessary. Therefore a list was implemented as the main carrier.

All information from one interpreter is stored in attribute–value pairs in one list. There is one list per sketch and, since multiple sketches from one interpreter are possible, one list per interpreter. Including the main information carrier list, a three-dimensional data structure is used to exchange information between the sketch interpretations (see Figure 9).

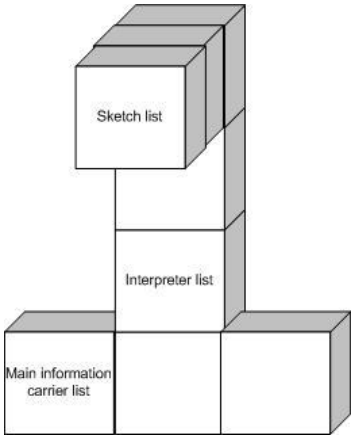

Figure 9. Three dimensional data structure which acts as an information carrier between the sketch interpretations

After the information storage system was designed and implemented, the information retrieval was implemented. Since an interpreter must search through all information in a list, it has to specifically know what to look for. Both information storage and retrieval were implemented in the ER and UI interpreters.

Using this new cross domain interpreter, a set of diagrams such as that in Figure 5 can be successfully interpreted to produce a MySQL database and Java UI. The generated UI is shown in Figure 6 including a table which shows the data retrieved from the database. This table is displayed by pressing the 'Receive' button on the UI. The process diagrams which contain the information about how to submit and retrieve data from the database are situated in the lower middle. The ability to successfully recognize a process diagram is not implemented in InkKit yet. However, for presentation purposes it has been assumed that InkKit could recognize them. It must be pointed out that while the processes are currently hard-coded, the information necessary for the sketch interpreters to communicate with one another is available.

6. Discussion

When systems are created the first step is often to outline the system's design and its capabilities. One frequently-used and efficient method of doing this is to sketch diagrams describing the single parts of the complex system. InkKit and other sketch tools already recognize different types of diagrams. However, all these tools are limited to recognizing and interpreting one diagram type at a time. By overcoming this shortcoming, new opportunities are created such as

automatically generated interpretation results where the different views interact with each other.

This demands an information exchange between the sketches when being interpreted. Since InkKit's modular design does not allow for direct coordination of this exchange, the different diagram plug-ins are required to do so. We found and explored several possibilities to organize an information exchange. One method is to use a communication protocol; another is to pass a data structure along the sketch interpretation. The latter has the advantage of being easier to extend and implement. However, it also has many disadvantages which a communication protocol would solve. The biggest drawback is the fixed interpretation order, which results in a sequential exchange schema that cannot be altered: if information is needed it has to be available otherwise the generated interpretation result is incomplete. Another problem solved by the communication protocol is the ability of the sketch interpreters to specifically ask for information. With the information carrier object approach all information has to be stored to ensure that it will be available when needed. The information carrier may also cause a problem of how to find the relevant information within the data structure.

Despite these drawbacks, the information-carrying object was implemented as it has the advantage of fitting more easily into InkKit's current architecture. We now have a better understanding of the requirements of the communication protocol which needs to be carefully designed before implemented. Any mistakes in its model would result in communication limitations, making careful planning necessary.

We simulate the existence of working process diagrams to demonstrate the new capabilities of InkKit regarding the information the diagram interpreters have about each other. Without the process diagrams no code defining the possible actions performed on the exchanged information would have been generated. In simulating the possible processes, we implemented only very general methods to perform on the information. These methods include the submission and retrieval of data entered in the UI to and from the MySQL database. Since the process diagrams describing data submission and retrieval are simulated their implementation was kept simple. For example, the database query used to request information does not include conditional statements even when information from multiple tables is requested.

Implementing a cross-domain information exchange between sketch interpreters into InkKit enables many new sketch tool features, such as letting the generated programs interact with each other.

7. Conclusion & Future Work

In this project we extended InkKit to recognize diagrams from different domains in a single step and connect the automatically generated preliminary diagram sketches into a specific data format. A "single step" means that the recognition, interpretation and output generation of all sketches in the active portfolio is computed in one run. Combining the representations results in independent software components from the various domains which interact with each other. Since sketch tools were already able to recognize different domains in separate steps, the next logical stage was to interpret sketches from different domains in one step and compute the relationships.

We explored different approaches to communicate between the sketches while they were being interpreted. We decided to use a three-dimensional list structure (see Figure 9) to provide this information exchange. This three-dimensional list structure was easier to implement and maintain than the alternative communications protocol.

In this project we realized the cross-domain interpretation between the UI and ER diagrams. Existing plug-ins in InkKit have been modified in order to be able to exchange information with other diagram interpreters. To make the information exchange more flexible, the three-dimensional database should be replaced by a communication protocol. This would lead to a new set of opportunities such as an information exchange which is independent from interpretation order, is more intuitive and easier to extend.

To assess the efficacy of these new capabilities a detail evaluation study is necessary. The focus of this evaluation should be on the new possibilities of collaborative work between experts from different domains.

8. References

1. Alvarado, C. and R. Davis, *SketchREAD: a multi-domain sketch recognition engine*, in *ACM SIGGRAPH 2007 courses*. 2007, ACM: San Diego, California.
2. Bailey, B.P. and J.A. Konstan. *Are Informal Tools Better? Comparing DEMAIS, Pencil and Paper, and Authorware for Early Multimedia Design*. in *CHI 2003*. 2003. Ft Lauderdale: ACM.

3. Bailey, B.P., J.A. Konstan, and J.V. Carlis. *DEMAIS: Designing Multimedia Applications with Interactive Storyboards*. in *ACM Multimedia*. 2001.
4. Chen, Q., J. Grundy, and J. Hosking. *An E-whiteboard application to support early design-stage sketching of UML diagrams*. in *Human Centric Computer Languages and Environments*. 2003. Auckland, NZ: IEEE.
5. Coyette, A., et al. *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*. 2004. Prague, Czech Republic: ACM Press.
6. Damm, C.H., K.M. Hansen, and M. Thomsen. *Tool support for cooperative object-oriented design: Gesture based modelling on and electronic whiteboard*. in *Chi 2000*. 2000: ACM.
7. Freeman, I. and B. Plimmer. *Connector Semantics for Sketched Diagram Recognition*. in *AUIC*. 2007. Ballarat, Australia: ACM.
8. Goel, V., *Sketches of thought*. 1995, Cambridge, Massachusetts: The MIT Press.
9. Hammond, T. and R. Davis. *Tahuti: A Geometrical Sketch Recognition System for UML Class Diagrams*. in *2002 AAAI Spring Symposium on Sketch Understanding*. 2002.
10. Landay, J. and B. Myers. *Sketching storyboards to illustrate interface behaviors*. in *CHI '96*. 1996. Vancouver, BC Canada: ACM.
11. Landay, J.A., *Interactive sketching for the early stages of user interface design*. 1996, Carnegie Mellon University: Pittsburg, PA.
12. Lank, E.H. *A Retargetable Framework for Interactive Diagram Recognition*. in *Proceedings of the Seventh International Conference on Document Analysis and Recognition - Volume 1*. 2003: IEEE Computer Society.
13. Lin, J., et al. *Denim: Finding a tighter fit between tools and practice for web design*. in *Chi 2000*. 2000: ACM.
14. Lutteroth, C., R. Strandh, and G. Weber, *Domain Specific High-Level Constraints for User Interface Layout.* Constraints, 2008. **13**(3).
15. Lutteroth, C. and G. Weber. *Modular Specification of GUI Layout Using Constraints*. in *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*. 2008.
16. Newman, M.W., et al., *DENIM: An Informal Web Site Design Tool Inspired by Observations of Practice.* Human-Computer Interaction, 2003. **18**(3): p. 259-324.
17. Patel, R., et al. *Ink Features for Diagram Recognition*. in *4th Eurographics Workshop on Sketch-Based Interfaces and Modeling* 2007. Riverside, California: Eurographics.
18. Plimmer, B. and I. Freeman. *A Toolkit Approach to Sketched Diagram Recognition*. in *HCI*. 2007. Lancaster, UK: eWiC.
19. Plimmer, B., G. Tang, and M. Young. *Sketch Tool Usability: Allowing the user to disengage*. in *HCI* 2006. London: ACM.
20. Plimmer, B.E. and M. Apperley. *Freeform: A Tool for Sketching Form Designs*. in *BHCI*. 2003. Bath.
21. Plimmer, B.E. and M. Apperley. *INTERACTING with sketched interface designs: an evaluation study.* in *SigChi 2004*. 2004. Vienna: ACM.
22. Plimmer, B.E. and M. Apperley. *Software for Students to Sketch Interface Designs*. in *Interact*. 2003. Zurich.
23. Rubine, D. *Specifying gestures by example*. in *Proceedings of Siggraph '91*. 1991: ACM.
24. Yeung, L.W.S., *Exploring beautification and the effects of designs' level of formality on the design performance during the early stages of the design process* in *Department of Psychology*. 2007, University of Auckland: Auckland.