Journal of Visual Languages and Computing 21 (2010) 98–108

Contents lists available at ScienceDirect

Journal of Visual Languages and Computing

journal homepage: www.elsevier.com/locate/jvlc

Generating systems from multiple sketched models

Paul Schmieder [a,*], Beryl Plimmer [a,1], Jean Vanderdonckt [b]

[a] Department of Computer Science, University of Auckland, Auckland 1142, New Zealand
[b] Université catholique de Louvain, Place des Doyens 1, 1348 Louvain-la-Neuve, Belgium

ARTICLE INFO

Keywords:
Sketch tools
Sketch recognition
Software modeling

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. Our extensions support software design processes by providing a sketch-based approach that allows the iterative creation of multiple outputs interacting with one another from the inter-linked models. We demonstrate that InkKit can generate a functional system consisting of a user interface with processes to submit and retrieve data from a database from sketched user interfaces designs and sketched entity relationship diagrams.

© 2010 Published by Elsevier Ltd.

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 in 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 computer-based sketch tools still lack in familiarity and intuitiveness of physical paper [1,2], 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 [3,4].

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 also recognize the sketches. Due to the diversity of possible sketches the demands on the underlying recognition 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 outline the design of a graphical user interface; the latter are used to specify database drafts, including the relationships between their entities. 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 combines different types of diagrams to generate a more complete model or system.

InkKit [5] is a software toolkit used to recognize and convert user-drawn sketches from several domains into other representations. Each domain consists of one

* Corresponding author. Tel.: +64 9 373 7599x89357; fax: +64 9 373 7453x85453.
E-mail addresses: psch068@ec.auckland.ac.nz (P. Schmieder), beryl@cs.auckland.ac.nz (B. Plimmer), jean.vanderdonckt@uclouvain.be (J. Vanderdonckt).
[1] Tel.: +64 9 373 7599x82285; fax: +64 9 373 7453x85453.

1045-926X/$ - see front matter © 2010 Published by Elsevier Ltd.
doi:10.1016/j.jvlc.2009.12.003

recognizer and multiple output modules. InkKit can recognize sketched diagrams from the same domain, which are split into several parts as shown in Fig. 1.

In this project we extended InkKit so that it combines the interpretation results of different types of diagrams. The generated software representations of the sketched diagrams can interact with each other because the necessary information is exchanged during the interpretation process. The contribution of this project is that this interaction enables many new sketch tool possibilities such as the generation of simple software systems rather than single individual system components. This allows the iterative development of low-fidelity sketches, which have been proven to be better for early stage design [1] and quick generation of operational prototypes.

Our exemplar is ER and UI diagrams which are recognized and used to generate a database and a connected user interface. Afterwards the user can enter data in the UI which is then transmitted to the database.

This method 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. Ideas can be explored and iteratively developed. Once they are finished, they could import their sketched ideas into one project and link the related parts which can then be further processed.

The structure of this article is as follows. The next section presents our motivation; this is followed by related work and a description of InkKit. We then describe the requirements for cross-domain recognition and our approach to implementing these requirements. The evaluation using the cognitive dimensions framework is described before a general discussion and final conclusions and future work is presented.

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 back 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 explore the intersection between tools, models and systems more easily.

There are various methodologies, models and diagrams used to describe software systems (for example UML). Bringing the different models together to describe a system is a well-known approach in software design known as model-driven architecture (MDA). At the most basic level, 'ordinary' software systems consist of a user interface, data and processes. Fig. 2 shows a simple set of three diagrams that could be used as a first iteration of

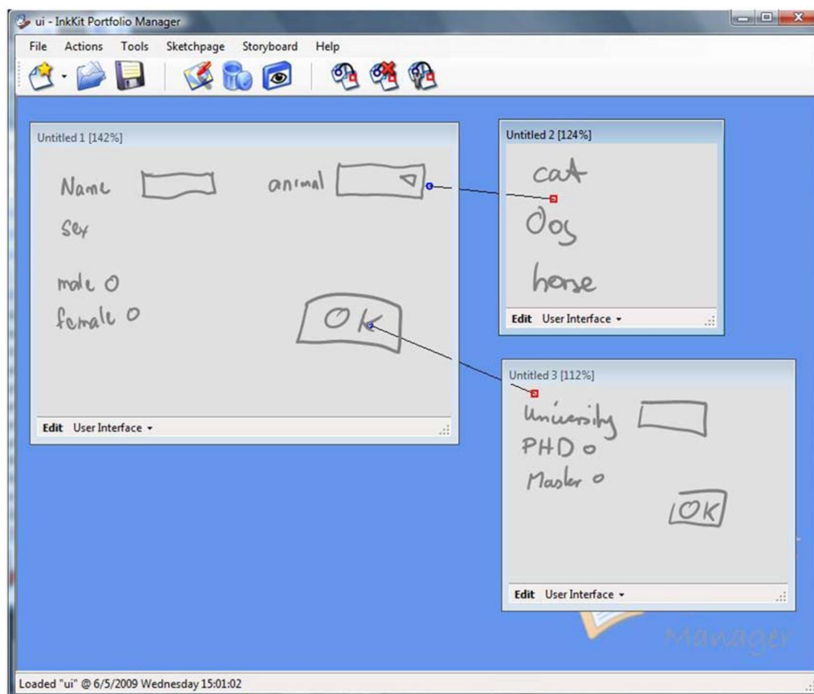

Fig. 1. InkKit portfolio manager which contains three sketch pages from the UI domain connected via rubber bands.

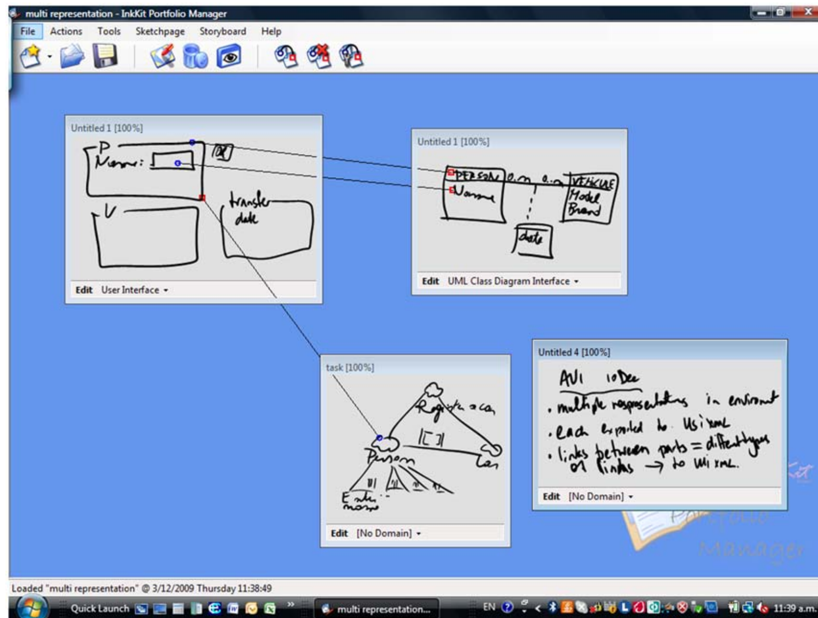

Fig. 2. Sketches of related user interface, class diagram and process hierarchy and the notes page.

a design. Our goal is to take a set of related diagrams like this and use them to generate the software system.

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 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. Generic recognizers are designed to recognize a range of different diagram types, while specific engines are tailored to one type of diagram.

The first published sketch tools had recognition engines dedicated to one particular domain. For example, Silk [6], as one of the first, was published in 1996 and was specifically designed to recognize UI diagrams. Four years later Knight [7] followed, and another two years later Tahuti [8] was designed to recognize UML class diagrams. More recent sketch tools are DEMAIS [9], Freeform [10], JavaSketchIt [11], LADDER [12] and SketchiXML [13], which all recognize UI diagrams.

Examples of sketch tools which have a generic recognition engine are Lank's framework [14], SketchREAD [15] and InkKit [5]. 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 [1,16–18], UML class [7,8] or circuit diagrams [15]. 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 Fig. 1) can be recognized and a unified representation can be generated as if it were drawn in one sketch. Denim [19] 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 pages 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 interrelationships.

4. 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 Fig. 1). This design is robust and well tested [20] and enables intuitive user interaction. In addition to basic functions, such as sketch page resizing, moving and zooming, connectors (called rubber bands) between the sketch pages can be added. They represent a directed relationship between the connected pages; the rubberband connects a shape on one sketch page with its associated content on another sketch page. The start point (indicated by a blue dot) of a rubber band is directly associated with the component it originates from (the drop down box and button in the left page of Fig 1), however, the end point (indicated by a red dot) is related to the entire sketch page at the end point. For example the left-hand sketch in Fig 1 shows two source points; one

from the drop down list to the list in the top-right sketch and the other from the 'OK' button to another user interface form.

For example, in Fig. 1 three sketch pages are connected. The top left sketch page includes a control for list of animal names, which will be added from the "connected" list on the top left page when both user interface diagrams are recognized and interpreted. The same applies for the third sketch page, which is connected to the first sketch page's OK-button. 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 [1,2]. 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 [22] as a part of the Java output.

Fig. 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 [23]. Those strokes recognized as letters are grouped into words and recognized by an independent text recognition engine.

Sketched shapes can consist of more than one stroke. In order to use Rubine's classifier [24] single stroke algorithm to classify each shape, the strokes that constitute one shape have 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 with a single composed stroke [25].

After being joined the shapes are recognized using Rubine's classifier. At this point during the recognition, there is no domain-specific knowledge available; drawing ink is classified as primitives (basic-shapes), such as circles and lines. Domain-specific shapes (complex shapes) consist of a set of basic shapes. For example, Fig. 4 shows the user interface component "Combo Box". Before Rubine's classifier is applied the "Combo Box" is decomposed into its basic shapes: rectangle and triangle. Then both basic shapes get recognized independently.

To recognize a shape Rubine's classifier [24] computes its feature vector. This vector is used in a linear evaluation function against an existing vector for every known basic shape class. While a shape can be every sketched component, a shape class describes all these components belonging to the same class or type such as rectangles and circles. These functions have been trained using about 10 to 15 predefined examples for every basic shape class to determine the weights over all features. The basic shape class with the highest computed value for its linear evaluation function is the one to which the sketched shape belongs. The basic shapes for the training of the linear functions are taken from a predefined set stored in a library which can be extended by the user.

After all the sketched strokes are recognized as basic shapes (except those recognized as text strokes), the results get handed over from the generic recognition engine to the domain-specific one. For the example in Fig. 4 (see bottom bar), which has been designated as a user interface by the user, the rectangle and triangle are known as such and handed over to the user interface interpreter that interprets them as a dropdown list.

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

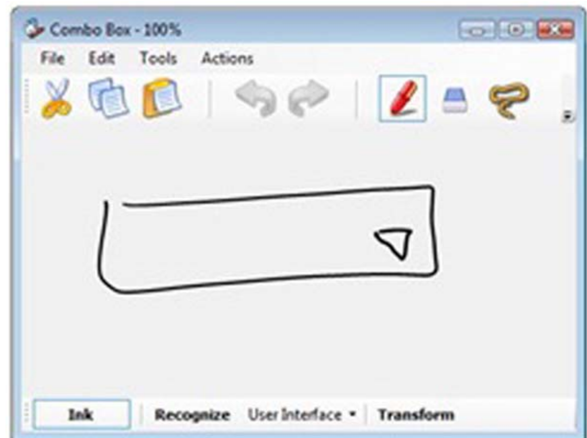

Fig. 4. A combo box consisting of the basic shapes rectangle and triangle.

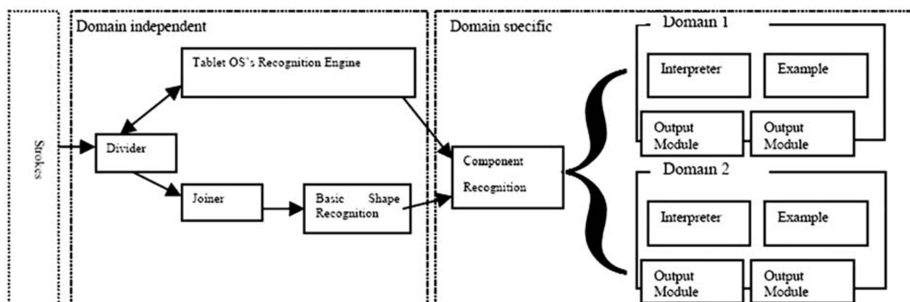

Fig. 3. Architecture InkKit.

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 interpreter 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 group to be part of a domain-specific complex component. Going back to the example in Fig. 4, at this point the triangle and rectangle are grouped together because of their spatial relationship; the rectangle encloses the triangle. Note that this new group is one of three groups, which are computed as each basic shape also forms an individual group.

Finally, using the likelihood calculation results of the basic shape groups, a hypothesis space is built which includes all these possible group combinations. 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.

For the example in Fig. 4 there are three groups at this stage of the recognition, the likelihood for each group being a complex shape is calculated. As the UI domain does not contain a component consisting of an individual triangle, the 'individual triangle' group gets the lowest likelihood. However, the UI domain contains complex shapes consisting of a rectangle (Text Box) and of a rectangle enclosing a triangle (Combo Box). The reason why the sketch gets classified as a "Combo Box" is that complex shapes consisting of multiple basic shapes are by default ranked higher.

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, sketched components 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 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. An output module maps the internal description of a diagram to a format specific representation such as Java or HTML. In the example shown in Fig. 4 the output module gets the information about a "Combo Box" being part of the diagram and generates an HTML file containing the "Combo Box". 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 that communicate through interfaces. A code segment is responsible for a specific task. This enables easy modification of the single layers to integrate and test new technologies.

5. Cross-domain requirements

To recognize the relationships between diagrams of different types and intelligently generate output that leverages these relationships a toolkit has to have certain capabilities. The most important properties are an interface supporting multiple sketches, a way to define relations between the different sketched diagrams, a recognizer and interpreter for at least one domain and some way of exchanging information between the diagrams during the recognition process.

5.1. Interface and relationships

Mechanisms which enable the definition of relationships between the different sketched diagrams must be provided. InkKit's well-tested rubber band visual approach [20] for pages of the same type of diagram can be extended to manage cross-domain relationships. The essential additional requirement is to be able to associate components on one diagram directly with components on another diagram. As an example Fig. 2 shows component-to-component relationships between the user interface and ER diagram components. This purely visual approach can leverage the start and endpoints as significant but does include cardinality information, which may be necessary in some circumstances. This cardinality information could be recorded by having different types of rubber bands or annotating the endpoints.

5.2. Recognition, interpretation and information flow

While the recognition process assigns the sketched components to their most likely predefined matches (basic shapes), the interpretation brings meaning to the overall sketch. For example, after the ER diagram shown in Fig. 5 is recognized, its components are known, i.e. the two entities, two attributes and three 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.

In InkKit [5] a diagram can be decomposed into several smaller diagrams, which aids overview and grouping. Until now all diagrams had to belong to the same domain because the information flow was bound to one

Fig. 5. Entity relationship diagram.

domain-specific interpreter. To recognize diagrams from different domains in one step every sketch has to be separately recognized whereas the interpretation is processed simultaneously. During the interpretation the single domain-specific interpreters have to exchange information such as component names and locations.

There are two main software engineering approaches to enable an information exchange: specifying and implementing a communication protocol, or providing a shared object acting as an information carrier that is passed to every interpreter.

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.

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 can result in several disadvantages. For example, every interpreter has to dump all information into the carrier resulting in a waste of memory. The reason to store all information is that the single interpreters do not know with whom they exchange information. Thus saving all information guarantees that all interpreters get the desired information.

However an information carrier also has a number of advantages including that it is 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.

6. 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, combining the data representation with the user interface. The processes 'submit' and 'retrieve' are hard coded to generate code specific representation (Fig. 6, lower two sketch pages) a functional system can be generating. InkKit's recognition engine has to be adjusted and an information exchange between the interpreters has to be established.

In order to interpret diagrams from different domains, the recognizers 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. After all sketches are recognized, they have to be interpreted.

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 rubber bands between the sketches (see Fig. 6). 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 [22]. ALM is focused on the tabstops between cells rather than on the cells of the grid like the Gridbag Layout Manager. This generalization of grid-based layouts makes ALM the most powerful manager in terms of adaptive layout resizing [26]. 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 easy interface to the Java front end that we planned. The implementation consists of 650 lines of code, making this module one of the more

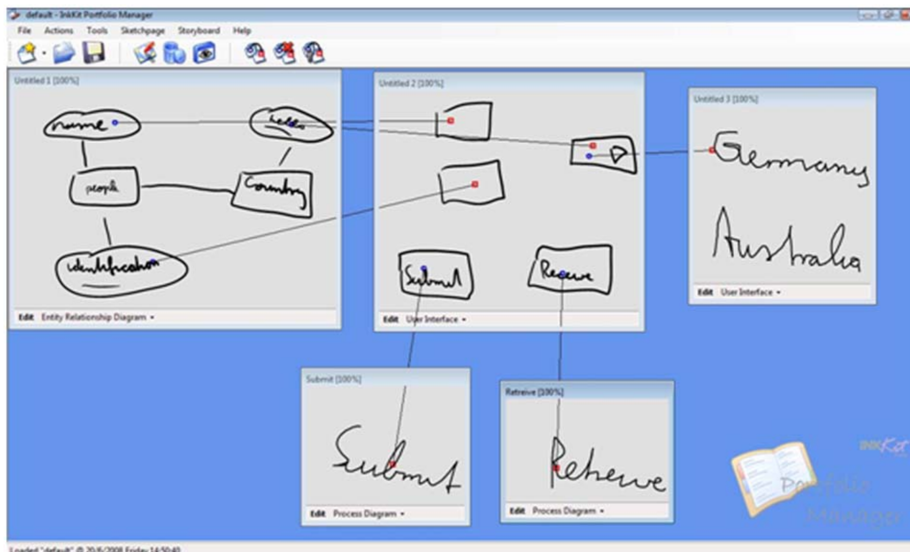

Fig. 6. InkKit's portfolio manager including a sketched ER diagram, user interface and processes.

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.

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. 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, hence the interpreters have to coordinate the communication by themselves. We considered two approaches to this, as described 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 dependencies between the sketches to inform the order in which the sketches should be recognized. The user can order the sketches manually using the GUI listing the required interpreters (see Fig. 8). When there are several sketches from the same domain (i.e. they have the same interpreter) they are ordered consecutively within the domain.

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, if desirable, the gaps in the generated output code can be manually completed by the user. This has the advantage of allowing immediate generation of partial or ambiguous designs that can be used experimentally using the design process.

Using this new cross-domain interpreter, a set of diagrams such as that in Fig. 6 can be successfully interpreted to produce a MySQL database and Java UI. The generated UI is shown in Fig. 7 including a table which shows the data retrieved from the database. This table is displayed by pressing the 'Retrieve' 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 of Fig. 6. A process diagram recognizer is not yet available in InkKit—for presentation purposes it has been assumed that InkKit could recognize them. However, while the processes are currently hard-coded, the information necessary for the process diagram interpreter to communicate with other interpreters is implemented and information between it and other interpreters is automatically exchanged.

7. Cognitive dimensions

We used Green and Petre's [27] cognitive dimensions (CDs) to evaluate the cognitive aspects of cross-domain diagramming. The recommended practice for conducting a cognitive dimensions evaluation is to describe the environment in which the interaction takes place and detect the system's notation [28].

Pen input is supported by a broad variety of devices such as Tablet PCs, Handhelds and special monitors. In terms of utilization the digital pen can be best compared

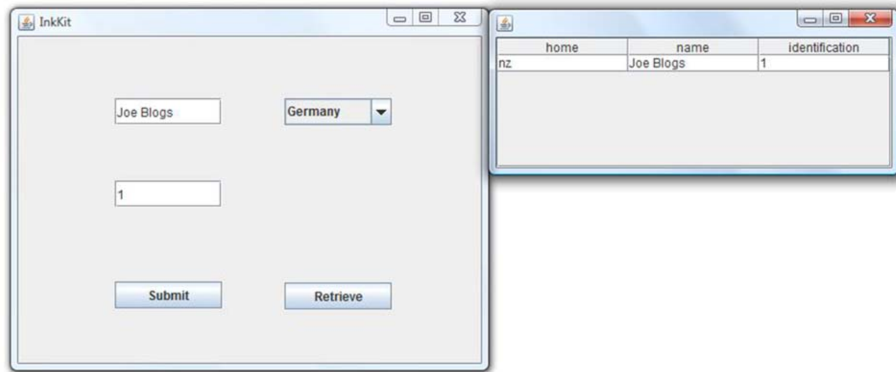

Fig. 7. User interface and database table automatically generated from the sketches in Fig. 6.

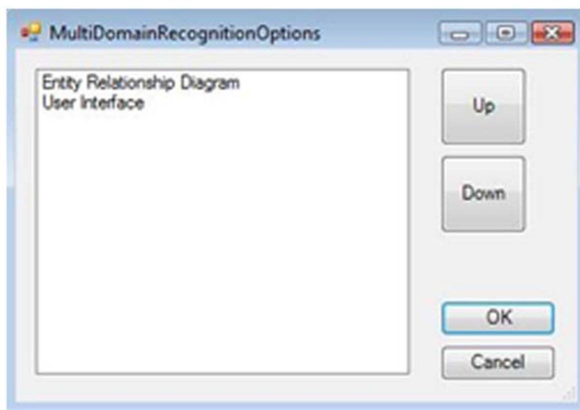

Fig. 8. Interface to order the interpreters.

to its non-digital equivalent. Both types of devices are used to record information in the form of writing or drawing on an appropriate canvas. Such a digital canvas is incorporated by our tool InkKit [5], which additionally provides methods to manipulate the drawn input, e.g. erasing, resizing, selection, redo/undo and cut.

In the following we provide a brief analysis of the CD's categories which were most influenced by our extension. The CDs which are discussed in this section have been chosen to outline the benefits of the new abilities to sketch software systems and develop them gradually. The focus specifically lies on the obvious as well as on the hidden parts this kind of progressive system development contains within itself. This becomes particularly evident when sketching as this kind of interaction is virtually free of constraints but also free of feedback; there is no obvious evidence as to whether things will go according to plan or not.

7.1. Premature commitment

There is no order forced on the user when sketching diagrams. The user is able to work on all sketches in an arbitrary order. Thereby the user can arbitrarily change between the different sketches. This switching between sketches and manipulating them in a constraint free manner is crucial to the progressive development of systems as the development steps can be arbitrary in size and order; every aspect of the system (distributed over the sketch pages) can be changed at will without any restrictions. When designing InkKit, this freedom of creativity was an important factor in order to achieve the overall goal of creating a sketch environment, which is as intuitive and easy to handle as possible.

7.2. Hidden dependencies

The dependencies between different domains are not visible to the user. It is possible that domains which are supposed to communicate (indicated by rubber bands) with each other cannot do so because of missing information. Due to the chosen structure, which handles the information exchange between the single domains the necessary information might not be available when needed. There is no feedback given as to whether the information exchange between the sketch pages works until the system is generated. Even then the user has to manually inspect the generated results to identify possible errors that occurred during the system generation. Due to the explorative nature of sketching and the proposed system, the user can 'play' around and change the order of sketch interpretations, which might result in a correctly generated system.

These current limitations can neither be influenced by the user nor actively avoided. It has to be pointed out that the system in its current status is still in development. As outlined before, the information carrier was chosen over a protocol to exchange information because of its flexibility and adaptiveness. These characteristics were very important when designing the extensions because the problem space is not well known and necessary changes should be as easy as possible to integrate.

7.3. Secondary notation

In order to support the users during the early design phase it is possible to add annotations to the sketches

which are not recognized. The purpose of those annotations is simply to aid the user in the process, which is of great importance to success as indicated by Myers et al. [29]. Annotations can be of arbitrary appearance because they do not have any notation constraints. Annotations have to be contained in sketch pages of type annotation. To indicate the association of annotations to sketches rubber bands can be used.

7.4. Viscosity

At all times sketches can be incrementally refined or modified; e.g. components can be easily added, deleted, selected or moved. The transcription of different sketches (to copy components from one sketch to another) is also supported. However, there is no automated mapping of the components to the new notation. This means that copied components belonging to one notation also have to be part of the target notation. If that is not the case the copied components will be incorrectly recognized.

7.5. Hard mental operations and progressive evaluation

The idea behind sketching is the ease and familiarity with which ideas can be visualized. This simple way of exploring the possibilities represented in the sketched diagram is proven to be more efficient in terms of producing ideas than widget-based tools [1,2]. One reason contributing to the efficiency is the nonexistence of hard mental operations necessary to express ideas. This is why the user can concentrate on the exploration of possibilities rather than on the process of putting a diagram together that represents the ideas. This whole idea is reflected in InkKit's interface: at any time the user can add diagrams, modify them and define relationships between them.

The same applies for progressive evaluation. At every stage the user can recognize all sketched diagrams including their relationships to check and explore the generated output. It is also possible to recognize one diagram rather than the entire set (when multiple sketch pages exist) to test whether this one diagram in isolation is correct.

As stated before, the basic principle of sketching is an easy and intuitive form of progressive evaluation. One might start drawing only the very essential components of a diagram and add components while confirming that the concept still works.

7.6. Closeness of mapping

To demonstrate the ability to recognize diagrams from different domains and connect the automatically generated output we used UI, ER and process diagrams. The idea is to sketch a set of related UI and ER diagrams, create relationships between the ER's attributes and UI's input components such as text boxes or combo boxes (see Fig. 6). There is a high closeness of mapping between the sketches and the automatically generated MySQL databases and Java UIs. To improve the aesthetics of the generated Java interface the Auckland Layout Manager (ALM) was used. Managing the output mainly affects the size and alignment of the UI components (see Fig. 7) decreasing the hard mental operations of users interacting with the generated interface.

The sketched process diagrams describe the actions triggered when clicking on the Submit and Retrieve buttons. In our implementation the sketch describing the process only consists of one signal word, i.e. either submit or receive. This highly abstract approach to visually represent processes results in a very low closeness of mapping. Regarding hard mental operations our approach of using key words to symbolize processes is extremely user-friendly due it's effortlessness to learn and facilitate.

8. 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 with 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 would be 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-carrier 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 implementation. 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 actions to perform on the exchanged information would have been generated.

The choice of notation for the different diagram types can be difficult. The notation used to sketch UI diagrams consists of well-known components, which look similar to their corresponding UI components resulting in a high closeness of mapping. The degree of abstraction is comparatively low because of the closeness of mapping; the generated result and the original sketch look almost identical. When choosing a notation for process diagrams the decision regarding the level of abstraction is not straightforward.

In InkKit's current implementation the two processes to retrieve and submit data to the database are indicated by key words rather than being properly described with sketches. This approach is abstract and cannot be manipulated or adjusted in any way. This is partly because there is no visual representation of processes which offers the means to sketch detailed enough processes; e.g. with a closeness of mapping close to the UI domain's one. The commonly known visual representations of processes (e.g. UML diagrams) are highly abstract and depict the process from different perspectives such as structure or information flow. Due to this abstract notation a lot of information has to be inferred resulting in high viscosity of a system; the system is highly resistant to change. High viscosity limits users in their design freedom by putting additional constraints on them, which goes against the purpose of our sketch toolkit as being intuitive and with natural interaction. Ideal would be a visual representation of the processes which is similar to the UI ones in terms of closeness of mapping and viscosity. To make a good decision regarding the process notation, which includes the abstraction trade-off, more research is necessary.

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.

Adding 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. UI and ER diagrams are good examples of related diagrams that have allowed us to explore many of the issues with cross-domain recognition. However there are possibly other requirements that different types of domain combinations would reveal. One which we are particularly aware of is the cardinality of relationships between the different sketches.

9. Conclusion and future work

In this project we extended InkKit to recognize diagrams from different domains in a single step and connect the automatically generated software of specific data formats. A "single step" means that the recognition, interpretation and output generation of all sketches in the active portfolio is computed in one run. The run combines the representation results in independent software components from the various domains with interaction between them. 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 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 list 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 and is more intuitive.

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

References

[1] B.P. Bailey, J.A. Konstan, Are Informal Tools Better? Comparing DEMAIS, Pencil and Paper, and Authorware for Early Multimedia Design. CHI 2003, ACM, Ft Lauderdale, 2003, p. 313-320.
[2] Yeung, L., Plimmer, B., Lobb, B., Elliffe, D., Effect of fidelity in diagram presentation, in: Proceedings of the 22nd British HCI Group Annual Conference on HCI 2008: People and Computers XXII: Culture, Creativity, Interaction—volume 1, United Kingdom: British Computer Society, Liverpool, 2008, p. 35–45.
[3] V. Goel, Sketches of Thought, The MIT Press, Cambridge, Massachusetts, 1995.
[4] Plimmer, B.E., Apperley, M., Interacting with sketched interface designs: an evaluation study. SigChi 2004, vol. Extended Abstracts, ACM, Vienna, 2004, p. 1337–1340.
[5] B. Plimmer, I. Freeman, A Toolkit Approach to Sketched Diagram Recognition. HCI, vol. 1, eWiC, Lancaster, UK, 2007, p. 205–213.
[6] J. Landay, B. Myers, Sketching Storyboards to Illustrate Interface Behaviors. CHI '96, ACM, Vancouver, BC Canada, 1996, p. 193–194.
[7] C.H. Damm, K.M. Hansen, M. Thomsen, Tool Support For Cooperative Object-Oriented Design: Gesture Based Modelling on and Electronic Whiteboard. Chi 2000, ACM, 2000, p. 518–525.
[8] Hammond, T., Davis, R., 2002. Tahuti: a geometrical sketch recognition system for UML class diagrams, 2002 AAAI Spring Symposium on Sketch Understanding.
[9] B.P. Bailey, J.A. Konstan, J.V. Carlis, DEMAIS: Designing Multimedia Applications with Interactive Storyboards, ACM Multimedia, 2001, pp. 241–250.
[10] B.E. Plimmer, M. Apperley, Freeform: a tool for sketching form designs, BHCI, Bath 2 (2003) 183–186.
[11] A. Caetano, N. Goulart, M. Fonseca, J. Jorge, Javasketchit: Issues in Sketching the Look of User Interfaces, AAAI Press, Menlo Park, 2002.
[12] Hammond, T., Davis, R., LADDER: a language to describe drawing, display, and editing in sketch recognition, IJCAI, 2003, p. 12–19.
[13] Coyette, A., Faulkner, S., Kolp, M., Limbourg, Q., Vanderdonckt, J., SketchiXML: towards a multi-agent design tool for sketching user interfaces based on USIXML, in: Proceedings of the Third Annual Conference on Task models and Diagrams, ACM Press, Prague, Czech Republic, 2004, p. 75–82.
[14] Lank, E.H., A retargetable framework for interactive diagram recognition, in: Proceedings of the Seventh International Conference on Document Analysis and Recognition-volume 1: IEEE Computer Society, 2003, p. 185–189.
[15] C. Alvarado, R. Davis, SketchREAD: A Multi-Domain Sketch Recognition Engine. UIST, ACM, Santa Fe, 2007, p. 23-32.
[16] J.A. Landay, Interactive Sketching for the Early Stages of User Interface Design, Carnegie Mellon University, Pittsburg, PA, 1996.
[17] J. Lin, M.W. Newman, J.I. Hong, J.A. Landay, Denim: Finding a Tighter Fit between Tools and Practice for Web Design. Chi 2000, ACM, 2000 p. 510–517.

[18] Plimmer, B.E., Apperley, M., Software for students to sketch interface designs, in: Rauterberg, M., Menozzi, M., Wesson, J., (Eds.), Interact. Zurich, 2003, p. 73–80.
[19] M.W. Newman, J. Lin, J.I. Hong, J.A. Landay, DENIM: an informal web site design tool inspired by observations of practice, Human-Computer Interaction 18 (2003) 259–324.
[20] B. Plimmer, G. Tang, M. Young, Sketch Tool Usability: Allowing the User to Disengage, ACM, HCI London, 2006, p. 164–167.
[22] C. Lutteroth, R. Strandh, G. Weber, Domain specific high-level constraints for user interface layout, Constraints 13 (7) (2008) 307–342.
[23] Patel, R., Plimmer, B., Grundy, J., Ihaka, R., Ink features for diagram recognition. 4th Eurographics Workshop on Sketch-Based Interfaces and Modeling Riverside, Eurographics, California, 2007.
[24] Rubine, D., Specifying gestures by example, in: Proceedings of Siggraph '91, ACM, 1991, p. 329–337.
[25] I. Freeman, B. Plimmer, Connector Semantics for Sketched Diagram Recognition. AUIC, ACM, Ballarat, Australia, 2007 p. 71–78.
[26] Lutteroth, C., Weber, G., Modular Specification of GUI layout using constraints, in: Weber G, (Ed.), 19th Australian Conference on Software Engineering, ASWEC, 2008, p. 300–309.
[27] T.R.G. Green, M. Petre, Usability analysis of visual programming environments: a 'cognitive dimensions' framework, Journal of Visual Languages and Computing 7 (1996) 131–174.
[28] Green, T. R. G., Blackwell A. F., 1998. "Cognitive dimensions of information artefacts: a tutorial." from 〈http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf〉.
[29] Myers, B., Park, S.Y., Nakano, Y., Mueller, G., Ko, A., How designers design and program interactive behaviors, VL/HCC 2008. Herrsching am Ammersee, Germany: IEEE, p. 177–184.