Seeking the User Interface

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI. 02912 USA
spr@cs.brown.edu

ABSTRACT

User interface design and coding can be complex and messy. We describe a system that uses code search to simplify and automate the generation of such code. We start with a simple sketch of the desired interface along with a set of keywords describing the application context. We then use existing code search engines to find results based on the keywords. We look for potential Java-based user interface solutions within those results and apply a series of code transformations to the solutions to generate derivative solutions, aiming to get solutions that constitute only the user interface and that will compile and run. We run the resultant solutions and compare the generated interfaces to the user's sketches. Finally, we let programmers interact with the matched solutions and return the running code for the solutions they choose. The system can be used not only for generating initial user interface code for an application, but also for exploring alternative interfaces and for looking at the user interfaces in a code repository.

Categories and Subject Descriptors

D.2.2 Design Tools and Techniques - *user interfaces.*

Keywords

Code search; user interfaces; user interface generation tools.

1. INTRODUCTION

User interfaces are always fun to design and create. The coding involves understanding complex widget sets, building multiple prototypes to try achieving the best user experience, convoluted, inverted-control-based code, and a variety of layout strategies. The resultant code is often complex, bug-ridden, difficult to maintain, and not particularly transparent. Testing user interfaces, especially during development, is difficult and time consuming; testing interfaces aesthetics and usability even more so. Yet user interfaces are a critical part of today's applications.

The goal of our research is to simplify and eventually automate the process of building user interfaces by letting the programmer rely on the growing repository of already developed and tested open source applications. Essentially we eventually want to use code search to generate the user interface. Programmers should simply sketch the user interface they want and then our tool will search the various repositories of open source applications, extract user interfaces from these applications, and return working code that is close to the programmer's design.

Open source code repositories and systems are growing exponentially. Ohloh now claims over half a million repositories with over 30 billion lines of code. (Last year it was 16 billion.) For many applications, already developed, tested, and used interfaces in the repository are similar to what a programmer is looking for and could be used with minor modifications.

Our work lets the programmer start with a sketch of the user interface along with some context information. We use the context information to search open source repositories for appropriate Java applications using existing search engines. We extract the user interface code from these applications, get the code to compile and run, check whether the generated interface matches the given diagrams, and let the programmer check the result interfaces by interacting with and editing them. The actual source code for the generated interfaces is returned to the programmer.

This approach can return interfaces that are fully developed, that include interaction code, code to handle different window sizes, and callback hooks. Such interfaces are more substantial than those generated by the user interface builders common to today's programming environments. The approach can also be used to explore the space of interfaces for an application domain, looking at different alternatives and filling in the gap between a preliminary sketch and a usable interface. Finally, the approach has been used to explore user interfaces as an aid to browsing code repositories.

The contributions of this work, in addition to showing the feasibility of using code search for user interface design, are:

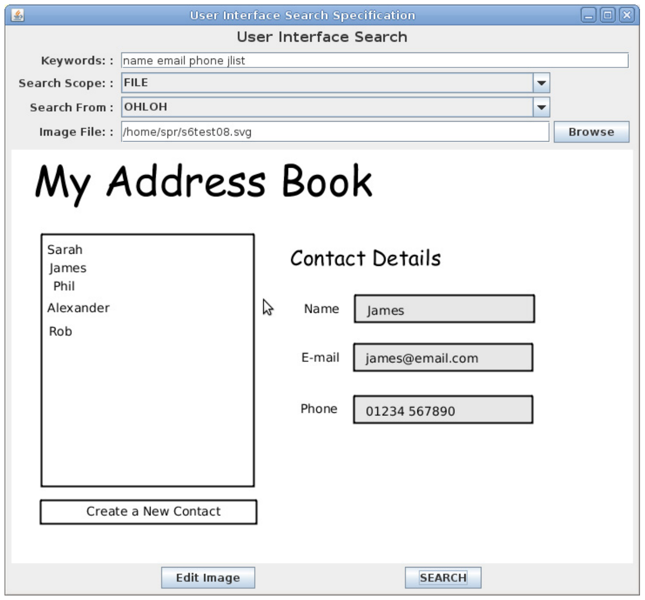
- A means for translating user interface sketches into a form that can be used to check if a given user interface is valid.
- Methods for gathering the appropriate code for a user interface from the simple results returned by code search engines.
- Transformations that take the raw code returned from code search, extract the user interfaces, and then make the code runnable outside of the original context.
- Techniques for matching a user interface sketch with an actual user interface.
- Tools that let the programmer see and interact with candidate interfaces to choose which they want the code for.

2. OVERVIEW

Searching for user interfaces can be broken down into three stages: specifying what to search for, generating candidate solution, and validating those solutions.

To specify a user interface, the user provides a sketch of what is desired along with a set of keywords describing the application context of the desired interface. Our tool is shown in Figure 1.

The user interface sketch is provided as an SVG file. While a freehand sketch might be more appropriate, we wanted to start with something more structured and slightly easier to interpret. SVG is a common standard, works well with the web, there are many available tools for creating and editing such diagrams, and Apache provides a suite of Java-based tools for SVG.



User Interface Search Specification

User Interface Search

Keywords: : name email phone jlist

Search Scope: : FILE

Search From : OHLOH

Image File: : /home/spr/s6test08.svg

My Address Book

Sarah
James
Phil
Alexander
Rob

Contact Details

Name: James

E-mail: james@email.com

Phone: 01234 567890

Figure 1. The user interface for specifying what to search for. The specification includes keywords and an SVG-based sketch. Search options include which code search engine to use and the scope of the search.

```
<COMPONENT HEIGHT='416.7938537597656' ID='U_70' TYPES='java.awt.Container' WIDTH='574 X='27' Y='1''>
  <COMPONENT DATA='My Address Book' HEIGHT='51' ID='U_51' LEFT='U_70' TOP='U_70' TYPES='javax.swing.JLabel' WIDTH='386' X='27' Y='15' />
    <COMPONENT DATA='E-mail' HEIGHT='10' ID='U_64' TYPES='javax.swing.JLabel' WIDTH='41' X='334' Y='233' />
    <COMPONENT DATA='Contact Details' HEIGHT='20' ID='U_60' TYPES='javax.swing.JLabel' WIDTH='171' X='321' Y='114' />
    <COMPONENT HEIGHT='32 ID='U_62' RIGHT='U_70' TYPES='javax.swing.JTextField' WIDTH='207' X='393' Y='166' />
    <COMPONENT HEIGHT='291' ID='U_52' LEFT='U_70' TYPES='javax.swing.JList,javax.swing.JTextArea,javax.swing.JEditorPane' WIDTH='245 X='34'
            Y='97' />
    <COMPONENT DATA='Name' HEIGHT='10' ID='U_61' TYPES='javax.swing.JLabel' WIDTH='39' X='337' Y='178' />
    <COMPONENT HEIGHT='33' ID='U_67' RIGHT='U_70' TYPES='javax.swing.JTextField' WIDTH='207' X='392' Y='283' />
    <COMPONENT BOTTOM='U_70' DATA='Create a New Contact' HEIGHT='29' ID='U_58' LEFT='U_70'
            TYPES='javax.swing.JButton,javax.swing.JMenuItem' WIDTH='250' X='32' Y='403' />
    <COMPONENT DATA='Phone' HEIGHT='11' ID='U_69' TYPES='javax.swing.JLabel' WIDTH='41' X='333' Y='292' />
    <COMPONENT HEIGHT='33 ID='U_65' RIGHT='U_70' TYPES='javax.swing.JTextField' WIDTH='207' X='392' Y='222' />
</COMPONENT>
```

Figure 2. Hierarchical component specification generated from the diagram shown in Figure 1. Each component includes a position and size.

When the user completes the specification and hits the search button, we build a Java code search request for a modified version of our S^6 search engine [36]. To do this we transform the user's sketch into a *hierarchical component description*. This description includes the components that should be in the user interface and the relationships among those components. Components can be nested. For each component, the description includes a set of Java Swing/AWT widget types that can be used to implement this particular component. The component description from the example shown in Figure 1 is shown in Figure 2.

Next, S^6 uses the keywords to find candidate solutions from an existing code search engine such as Ohloh, Krugle, or Github. S^6 next looks for candidate user interfaces. A candidate solution can be a class that implements a Swing/AWT component or a non-private method that returns such a component. Next S^6 applies transformations to each solution in an attempt to create code that is compilable, runnable, and only contains the user interface. The result of each transformation is a new solution that can also be transformed. The end result of these transformations is a set of candidate user interfaces that might meet the user's criteria.

We validate these solutions in several ways. First, we ensure the code compiles and runs. Second, the user interface generated by the code needs to match the hierarchical component specification. Third, the interface needs to look and act correctly.

For the first two of these, S^6 compiles and runs the code, and then matches the user interface generated in the run against the component specification. The various constraints and values included in the specification are used to generate a matching score which is used to rank the solutions.

The task of seeing whether the user interface is appropriate and what the programmer was thinking of is left to the programmer. Our tool presents the candidate solutions to the user first by showing images of the interface as seen in Figure 3. The user can accept or reject the solutions directly, based on their image. Alternatively, if a solution is clicked on, then the system will run the user interface along with a viewer that lets the user investigate the widget hierarchy and the various events generated by interaction.

Once the user has selected a set of acceptable solutions, they hit the "Show the Code" button to get a display of the resultant code. This is shown in Figure 4.

In the next section we describe S^6 and other related work. Section 4. describes how we generate the hierarchical component description from the SVG diagram. Section 5. describes the various transformations and other extensions to S^6 that are needed for handling user interfaces. Section 6. describes the matching algorithm along with the tools and techniques for presenting the solution to the user. Section 7. describes our experiences to date and offers an evaluation of the work. Section 8. then concludes by describing our on-going work.

3. RELATED WORK

Creating graphical user interfaces has been a difficult problem since the 1980's when such interfaces starting to become common.

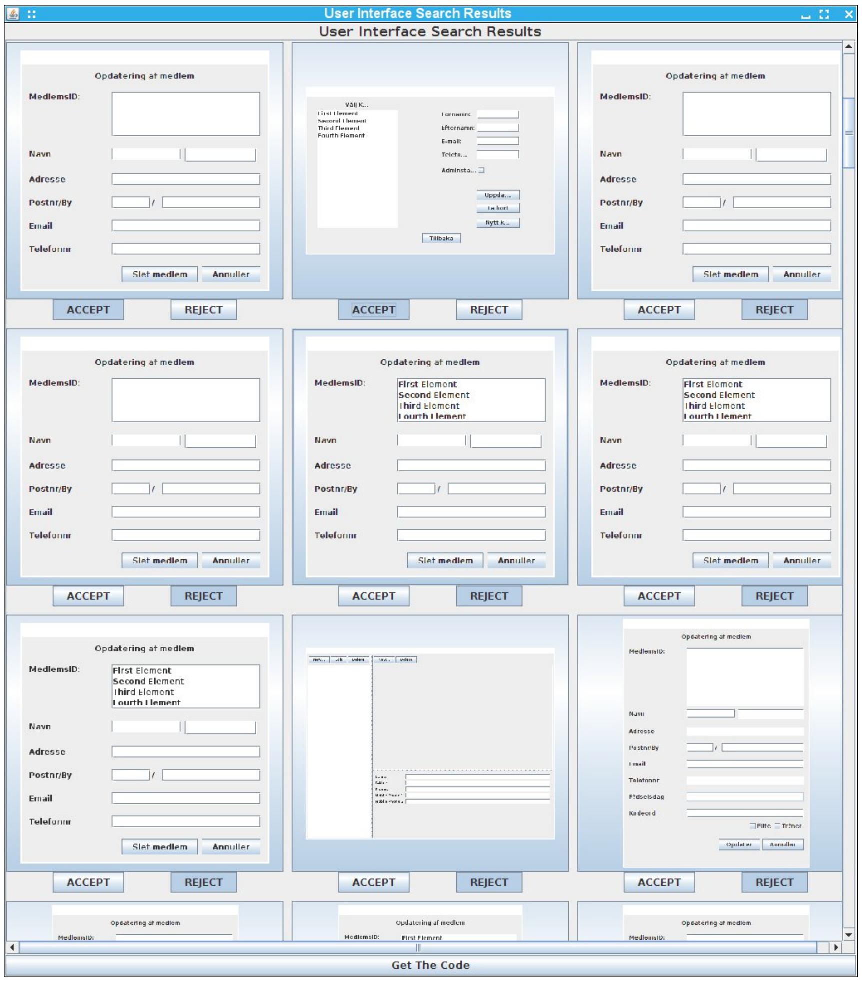
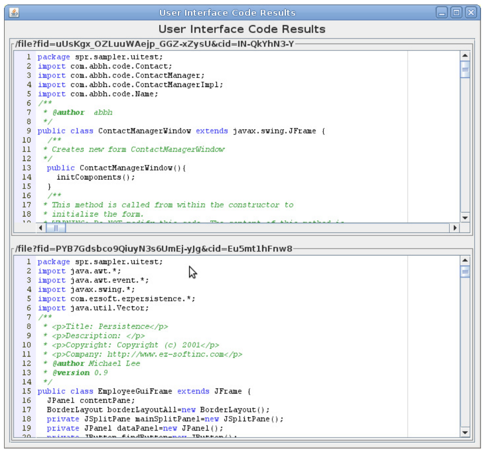


Figure 3. The resultant display showing potential solutions for the address book sketch of Figure 1. Each solution can be accepted or rejected by the user. Moreover, the user can experiment with the solution by clicking on it.



/file?fid=uUsKgx_OZLuuWAejp_GGZ-xZysU&cid=IN-QkYhN3-Y

```java
package spr.sampler.uitest;
import com.abbh.code.Contact;
import com.abbh.code.ContactManager;
import com.abbh.code.ContactManagerImpl;
import com.abbh.code.Name;
/**
 * @author  abbh
 */
public class ContactManagerWindow extends javax.swing.JFrame {
  /**
 * Creates new form ContactManagerWindow
 */
  public ContactManagerWindow(){
    initComponents();
  }
  /**
 * This method is called from within the constructor to
 * initialize the form.
```

/file?fid=PYB7Gdsbco9QiuyN3s6UmEj-yJg&cid=Eu5mt1hFnw8

```java
package spr.sampler.uitest;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import com.ezsoft.ezpersistence.*;
import java.util.Vector;
/**
 * <p>Title: Persistence</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2001</p>
 * <p>Company: http://www.ez-softinc.com</p>
 * @author Michael Lee
 * @version 0.9
 */
public class EmployeeGuiFrame extends JFrame {
  JPanel contentPane;
  BorderLayout borderLayoutAll=new BorderLayout();
  private JSplitPane mainSplitPanel=new JSplitPane();
  private JPanel dataPanel=new JPanel();
```

Figure 4. Final display showing the code for the user interfaces the user accepted.

While most interfaces then and now are still hand-coded, there have been a wide variety of tools developed to assist and even attempt to automate the process. A good summary of the state of the art in 2000 is provided by [29].

3.1 User Interface Generation

Modern development environments such as NetBeans, Visual Studio, and Eclipse support user interface generation. They let the developer drag and drop widgets into containers and to set the various properties of the widgets. Once a user interface is designed, the basic code for the interface can be generated. The programmer can modify this code to interact correctly with other portions of the application. These tools provide some simplification of the process, but are not ideal in that a) they don't handle interaction, data validation, or other interface dynamics; b) they often use absolute positions and it is complex to generate easily resizable interfaces; c) they don't handle dynamically generated interfaces where the interface depends on external files or the state of the application; d) the code that is generated may not be in a style or form the programmer desires; and e) once the code is modified it becomes difficult to use the support to update or change the user interface. The latter is a problem because user interface design often involves the rapid iterative design, exploration and comparison of different interface implementations [16].

There are some tools that attempt to generate user interfaces without actually writing code. Some of these involve using non-procedural specifications such as Mozilla's XUL [12]. Others involve developing various models representing the underlying data and the presentation and then generating the interface from these models [27,33,41]. The model driven tools have been more successful when applied to specialized environments [14,30]. There has also been work on automatically adapting user interfaces based on device or user constraints [11,31].

3.2 Basic Code Search

Code search technology has been developed mainly as an extension to the very effective uses of web search in general. A variety of code search tools have been developed by researchers to help the programmer find the particular code they might be interested in out of the large available body of code [1-6,8,10,15,17-22,24,25,28,32,39,40,42-48]. This is in addition to commercial tools such as Ohloh (code.ohloh.net, formerly Koders at koders.com), Krugel (krugel.com), Github (www.github.com) and the now-defunct Google code search.

While code search shows much promise, it has not caught on extensively. To some extent, this is because the various code search engines are not particularly effective. However, even with an effective search, the programmer still has to do a significant amount of work in order to use the result. This includes checking whether the code actually does what is desired, adapting the code to their particular project, possibly debugging the code, and converting the code to their style and formatting standards.

3.3 Semantic Search with S^6

Our prior work on code search is the semantic code search engine S^6. S^6 attempts to address several of the problems with current code search technology by effectively automating the multiple tasks the programmer has to do manually in order to use the output of a code search tool [36,37].

S^6 can be used to search for either Java classes or methods. It provides a web-based interface that asks the user to first provide a description of what is wanted in terms of keywords and the semantics of the target code. The latter includes the signature for the target class or method, one or more test cases, and optionally contracts (preconditions and postconditions) and security specifications (e.g. the returned code should not do any file I/O).

Once this data is entered, S^6 processes the request. It first uses the keywords with an existing code search engine (Ohloh, Krugle, Github, GrepCode, or Sourcerer [2]) to get a starting set. It generally takes the first 100-200 files from the search results to build an initial set of solutions. The next step is to apply transformations to each solution to generate new solutions. This is done repeatedly until no more transformations are applicable and no new solutions are generated. These transformations include relatively simply ones such as change the name of the method to match the name in the specified signature or reordering the parameters; moderately complex ones such as replacing a parameter with an appropriate assignment; and complex ones such as extracting functionality from a method by finding a top-level statement computing a value of the return type, doing a backward slice of the code until the only free variables are of the parameter types, and then extracting the resultant code into its own function.

The system next takes all the resultant candidate solutions and does a dependency check. This check adds other code fragments such as field declarations and auxiliary methods from the initial file that might be needed to make the candidate compile. It removes candidate solutions with unmet dependencies that will not compile. For each passing candidate the system generates a test program that tests that candidate against the user’s original test cases, contracts, and security constraints. This test program is compiled and run using Apache Ant and JUnit. The system does an additional pass looking at the output from the tests, and will try additional transformations as appropriate, for example, transformations that handle off-by-one or uppercase/lowercase errors.

Finally, the system takes the candidate solutions that pass all the test cases and passes the resultant code back to the user. It gives the user the option of different formatting styles [35] and different orderings for the results (e.g. fastest to slowest, smallest to largest, least to most complex). It also provides license information for each of the fragments. The user can then take the result, cut and paste it into their program and use it with the confidence it actually compiles and passes their test cases.

S^6 provides a general framework for using code search for different purposes. It starts with keywords to identify a set of initial candidate files. Next, it uses a set of transformations that convert these candidate files into initial candidate solutions. Next it transforms the candidate solutions so that they are likely to compile and run. These transformations are limited by applying an intermediate check as to whether the solution is feasible or not. Finally, it needs to compile and validate the resultant solutions. Our search tool implements and specializes this framework for user interfaces.

3.4 Other Search Tools

Other code search tools use test cases rather than keywords and are closer to the approach taken by S^6. A recent test-driven approach is CodeGenie [22]. More recent code search work on test cases includes [23]. Test cases and semantics have also been used in a similar fashion for finding web services [9,34].

Most current research code search tools are based on information retrieval techniques. Early work here demonstrated that keywords from comments and variable names were often sufficient for finding reusable routines [10,25]. Later work here did query refinement either directly [43], by looking at what the programmer was doing [48,49], using an appropriate ontology [47], using learning techniques [8], using natural language [7], or using collaborative feedback [46]. Recent approaches, such as Assieme [18], Sorcerer [2], Codifier [3], Exemplar [13] and Portfolio [26] incorporate program structure and semantics as a search basis.

4. SPECIFYING USER INTERFACES

Our goals in specifying a user interface for code search were threefold. First, we needed to provide an appropriate starting point for the search process. Second, we wanted to use a natural metaphor, starting with sketches as designers typically do. Third, we wanted to be able to check the result against the specification so we could test if a generated search solution was appropriate.

There are two aspects to identifying a starting point for the search. The first is the set of keywords that will be used in conjunction with an existing code search engine to find initial files. The second specifies if the solution should be within a single file, within a single package, or spanning multiple packages.

User interfaces can be implemented in a variety of ways. Simple interfaces and interfaces developed using user interface builders are often implemented within a single file. More complex interfaces, where the user creates custom components, uses custom models for tables or lists, or implements complex internal functionality, are often implemented in multiple classes within a single package. Applications that have multiple user interfaces may use a common user interface package for support code while implementing the actual interface in a separate package within the system.

In order to accommodate these different user interface implementation styles, we support initial solutions that are either file-based, package-based, or system-based. For package based solutions we start with the initial file returned from the code search engine, search for other classes in the same package, and merge the results into a single virtual file for further processing. This merger yields a single Java file containing multiple classes that would typically not compile directly. However, we retain enough information to separate this into multiple files when we need to compile it. The merger also takes into account the different imports for the different files, yielding a common set of imports by replacing simple names with qualified names where necessary.

In the case where the user interface might span packages, we start with the initial file, add the other files for the package as above, and then use the import clauses and qualified names in the result to identify packages that share a common prefix with the original one. All the classes in these packages are merged with the original file as well and the process is repeated until no new packages are identified. The merging here moves all the files to a single package, updating names and import statements accordingly.

The remainder of the specification is a sketch of the desired interface. While we would ideally allow free-form input, this seemed overly complex for an initial system. Instead we assume that the user creates the sketch using an SVG editor. SVG is web-friendly, matching the current S^6 web interface. Moreover, there are several tools available for creating and editing SVG files such as Inkscape and the web-based svg-edit. The edit image button on the bottom of the interface will bring up an appropriate editor, either Inkscape if it is installed on the system or GLIPS Graffiti. Finally, the search button at the bottom right of the interface starts the whole user interface search process.

The SVG-based user interface sketch addresses our second criteria, letting the user start with a sketch.To make this usable by S^6 we analyze the sketch and translate it into a hierarchical component description in stages.

The first stage finds potential components. We use Apache's Batik package to map the SVG diagram into drawable components, either shapes (rectangles, rounded rectangles, ellipses or general paths) or text. We further characterize shapes as either boxes, input regions, lines, symbols, icons, rounded regions, or text. Input regions are boxes that are either lightly filled or that have a thicker than normal border. Lines are either lines or are boxes that have essentially one dimension. Symbols are shapes that are small and can represent either a simple button (e.g. a radio button), an icon, or an arrow (for a scroll bar). Icons are larger two dimensional symbols and can represent larger icons or general drawings. Squiggly paths that are long and narrow are taken to represent potential text. Text regions are further characterized as containing single or multiple lines.

The next stage creates a hierarchy of the candidate components. This is done by looking at the bounding boxes of each component and seeing what other components are nested inside. Here we use an approximation to actual nesting to accommodate minor errors in the sketch. For example, if a string happens to lie mostly inside a rectangle, but extends outside slightly, we consider the string to be nested. Once we determine all nestings, we build a hierarchy by finding the innermost nesting for each component. Finally, if there is no unique top-level component, we create one.

The next stage attempts to merge logical groups of components and to characterize the components so that we can assign potential widget types for checking. This is done with a series of hand-coded checks that assign properties to the components and clean up the hierarchy. The actual checks done here include:

- Looking for components containing only text subcomponents. This characterizes components as buttons if the text is a simple string and is generally centered or if the enclosing region is circular; or as single line or multiple line text (input) regions otherwise. If text is present we check for asterisks to indicate a password or hidden field and for only numbers to indicate a numeric field. Where components are further characterized, the text subcomponents are removed from the result hierarchy.
- Looking for combo boxes (buttons with a choice of options). These are text boxes with a symbol on the right. If one is found, the symbol and text are removed.
- Looking for toggle buttons such as radio buttons and check boxes. These are buttons or text components with an adjacent symbol. Where these are found, a new component is created spanning both the original components which are removed.
- Looking for menu and tool bars. These are long, narrow regions containing a one or more symbols, buttons or text strings.
- Looking for tables, trees and lists. Tables are characterized as boxes containing both vertical and horizontal lines and possible text elements. Lists can contain horizontal lines or multiple text items. Trees can contain vertical lines and have text areas that are properly offset. Any internal subcomponents are removed.
- Looking for scroll bars. These are either long or narrow regions that contain symbols at the top and bottom and possibly a box or symbols in the middle. Any internal symbols and boxes are removed if a scroll bar is identified.
- Looking for spinners. These are numeric fields with one or two symbols immediately to the right.
- Looking for sliders. We look for a long narrow component with additional symbols on top of it and with potential text immediately above or below. If a slider is identified, all the internal components are replaced with a single slider component.
- Looking for drawing areas. These are characterized as a component containing multiple symbols and shapes but no buttons.

The checks here are designed to be forgiving in order to accommodate minor errors in the original sketch. This comes first from the fact that the hierarchy determination is not strict. Moreover, the checks for aspect ratio accept an overly broad range of values; checks for horizontal and vertical lines allow ease; and extra marks or boxes that are small or seem irrelevant are ignored.

The fourth stage of component processing takes the resultant set of components and computes a set of relative positional constraints that can be used for checking. Each component can identify another component that is immediately above it, one that is immediately below, one to the left, and one to the right. Nested components can also be assigned a position relative to their parent, for example, a component that is at the top of its parent has the parent identified as the component immediately above it.

The final stage assigns potential widget types to each component. This uses the properties set by the above drawing analysis to create a list of candidate AWT/Swing widgets for each component. This is the only part of the specification stage that is dependent on the user interface being generated for Java AWT/Swing.

The resultant component hierarchy for the input shown in Figure 1 is shown in Figure 2. The box on the left is identified as either a *JList*, *JTextArea* or *JEditorPane*; the three boxes on the right are identified as *JTextField*s, and the box at the bottom as a *JButton* or *JMenuItem*. The remaining elements are either JLabels or the outermost Container.

Our user interface for specifying what to search for can be seen in Figure 1. The top three boxes define the starting point for search. The top box contains the keywords; the second box identifies the type of search; the third box selects the search engine to be used. The sketch, selected from a file, is displayed below.

5. GENERATING SOLUTIONS

To do the actual work of searching for a user interface, we substantially modified the S^6 search engine. The modifications generally fall into three categories. The first is handling packages and systems rather than individual functions or classes. These were described in the previous section. The second involves restricting the code to that relevant to the user interface by eliminating unnecessary elements. The third involves getting the resultant code to compile and run, effectively duplicating what a programmer might do when extracting the interface from the code.

S^6 for user interface search starts with the code generated from either files, packages, or multiple packages based on the initial code search. Each of these code files (with the latter ones being considered single files after all the code has been merged), is considered a candidate solution.

The next step is to identify potential user interfaces in each solution and generate separate solutions for each. This is done using S^6 code transforms. We first convert the package name to a standard one for the user interface. Next we find all candidate interfaces. These are non-private constructors for any class that extends *java.awt.Container* and any non-private methods of a class that return an object that extends *Container.* For each such candidate, we create a new solution by creating a new class with a standard name that either calls the appropriate constructor or first builds the class and then calls the identified method. Where there are multiple possible constructors, we generate separate solutions for each, using logical default values for any parameters. Similarly, if the identified methods take parameters, we will generate separate solutions using different default values for those parameters.

Each potential user interface solution is restricted by a transformation that eliminates any code that cannot be reached from the class added for the solution. The result of this is a set of candidate solutions that implement a potentially relevant user interface and that are restricted to the code needed for that interface.

The next step uses existing S^6 transforms along with Swing-specific transforms to take these solutions and build new solutions that have a greater possibility of compiling and meeting the user's needs. The transformations that we have added for handling multiple classes, user interfaces and Swing include:

- Removing any code that references undefined types or variables. If the code can't be directly removed (for example a return statement at the end of a method), the undefined value is replaced with a default value, with both null or 0 and a non-null or positive value being tried. This transformation also removes empty statements and private methods that have no remaining statements.
- Finding variables that are used before they are assigned to and adding initial assignments to their declaration. Such instances often arise because the assignment was previously removed.
- Cleaning up the class structure. This includes making inner classes be standalone classes, adding additional fields and changing the constructor; removing unneeded implements clauses; and merging a subclass with its superclass to form a standalone class.
- Mapping calls that use the *ResourceBundle* interface to use our own version that returns reasonable default values for each call. Also, handle calls to *Properties.load* appropriately. This handles many of the cases where the code has been internationalized.
- Repairing calls to AWT/Swing methods that have undefined or unusable parameters. This includes calls that set the text for various types of text widgets and calls that reference external files, for example calls that construct images and icons. The transform will replace any undefined string parameter with a unique string, will replace integer and Boolean parameters with a logical value, will replace color parameters with a valid color, and will replace images and icons with a known image or icon. Other parameters are replaced with either null or with a new instance of an object of the appropriate type. These transformations attempt to maintain the original interface in the face of computed or external values.
- Replacing anonymous classes that inherit from or implement a Swing or AWT interface that have undefined symbols in them with versions that will compile.
- Replace list, tree, and table models offered by the user with simple internal models. Many of the compilation and run time problems we encountered in attempting to use external solutions arose because the code attempted to use incomplete or unavailable models since the models themselves are integrated with the application and not the user interface, might not be present in the selected files, or might need to be generated from other sources such as a database or external file.
- Removing or replacing code that would cause the user interface to hang or become untestable. For example, if the application attempts to run a modal dialog, our test code will never have access to the result and the test will fail because it took too long.

To keep the number of solutions reasonable, the system applies a filter that eliminates solutions that cannot be transformed to match. In the case of user interfaces, it checks to see if the code has a reference to one of each of the sets of types needed by each of the user-specified components. This reference can either be direct or indirect (i.e. might be to a subclass rather than the class itself). If there is some user component that cannot possibly be implemented by the solution, the solution is discarded. The check ignores labels since these are not critical to the resultant interface.

The number of candidate solutions can vary considerably, but generally doesn't become excessive. For example, the search involved with Figure 1 considered 116 files derived from the Ohloh search engine, generated 236 initial user interface solutions, and found 569 solutions to test out of a total of 4,122 that were generated by the various transformations, and tested the first 500 of those to produce the results shown in Figure 3. The ordering of solutions to determine which to test is a part of S^6 and is a function of the initial rank returned by the search engine, the number of transforms done, and a random value to encourage breadth.

6. VALIDATING SOLUTIONS

The next step involves testing whether the code that was extracted as a potential user interface solution actually matches the user's sketch and meets their needs.

We take a two-step approach here. First, we match the generated user interface against the user's sketch. This match first checks that all the components of the user's sketch appear in the generated interface. If they do, then the match computes a score describing the quality of the match. The second step is to present the interfaces to the programmer, first by showing a screen shot of the interface, and second by actually running the interface and letting the programmer interact with it, explore its widget hierarchy and callbacks, and do some simple editing.

To match the generated user interface against the user's sketch, we run the generated solution and investigate the widget hierarchy that results. The code generated for each potential user interface solution returns a user interface object (instance of *java.awt.Container*) from which we extract the hierarchy using the basic methods of *Container.*

In addition to looking at the hierarchy, we ensure that the display is runnable and supports interaction. This includes putting non-window widgets inside a frame and ensuring that dialogs are non-modal. It also involves determining and setting a reasonable size for the resultant window, checking if the window can be resized and making sure the top level user component is visible.

Both the generated widget hierarchy and the user's hierarchy are trees and we use a modified form of tree matching to compare the two. The comparison is loose in that the generated hierarchy is likely to have many additional components and hierarchy levels. For example, it might be organized as multiple panels to effect a better layout; a widget might be contained in a scrolled region (which adds the scroll pane, the viewport, the scroll bars); or the top level might be a root pane with all its associated components. In addition, the actual implementation might include additional widgets that the user's sketch didn't account for. For example, in the address book example, there might be additional fields (e.g. telephone or office address) that other implementations included but the user hadn't thought of (and might want). We also allow a little leeway in the match by permitting a small set of original components (one or two, depending on the total number of components), to not be matched explicitly.

The tree matching we do effectively considers all logical assignments of the user specified components to actual widgets in the implementation. Matches need to satisfy four criteria:

- Each component has to be matched with a widget. This constraint can be relaxed to allow a small number of non-matched components.
- The top-level component needs to match the top-level widget.
- The widget matched with the component must be an object of a class that is either one of the types associated with that component (in the last stage of the user interface specification), or must be a subclass of that type. Java reflection is used to check subtypes.
- The hierarchy specified by the user's components must be reflected in the widgets. If component A is a child of component B in the specification, then the widget associated with A must be a child, either directly or indirectly, of the widget associated with B.

Because there can be an exponential number of matches (consider 20 user labels that can match 20 actual labels), the search is designed to find a reasonable match fast and will stop once oa maximum number of solutions (currently 1000) have been found.

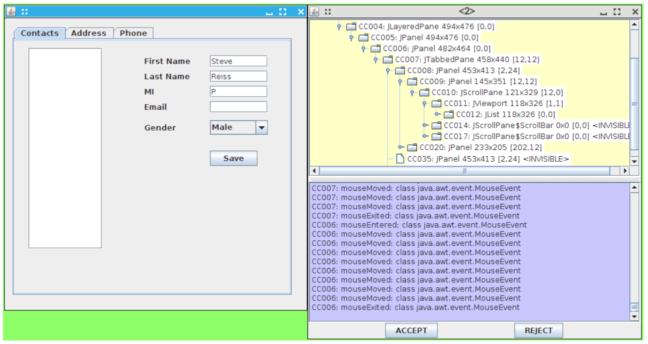
Once a match is found, we compute a heuristic score for that match. For each specified component this score takes into account

- Whether the component matched. The score is increased by 200 if so.
- How close the width and height of the component matches that of the widget. For both the width and height, if the actual value is within 100 of the user sketch value, the score is increased by 100 minus the delta.
- If text is associated with the component, the editing distance of that text versus any text associated with the wizard. Here we use reflection to call the *getText* method of the widget. The score is increased by a value between 0 and 100 depending on the quality of the match and the length of the text.
- If left, right, top, or bottom positional constraints are specified for the component, the distance in the implementation between this widget and the widget associated with the constraint. For each specified relationship, if the actual widgets are within 10 pixels, the score is increased by 50.
- Actual components that are not matched by a widget are penalized. We subtract 20 from the score for each extra label, 40 for each extra button or combo box, and 60 for each text field, list, table or tree.

The scoring tries to take into account the relative importance of each factor in assessing the match. It is designed so that obvious matches will have the highest score and be shown to the user first. Because the number of matches to date has not been excessive, the particular values chosen for scoring are not that important.

The next step is to get the user's opinion and validation for each of the matched interfaces. For each solution that matches the specification, S^6 creates two results for further matching. The first is an image of the widget as a PNG file and the second is a runnable JAR file that can be used to explore the widget. These are passed back to the front end along with a unique identifier and the score for each solution.

The interface for asking the programmer about the interfaces is shown in Figure 3. The programmer is shown the static images of



First Name: Steve
Last Name: Reiss
MI: P
Email:
Gender: Male
Save

```
CC004: JLayeredPane 494x476 [0,0]
  CC005: JPanel 494x476 [0,0]
    CC006: JPanel 482x464 [0,0]
      CC007: JTabbedPane 458x440 [12,12]
        CC008: JPanel 453x413 [2,24]
          CC009: JPanel 145x351 [12,12]
            CC010: JScrollPane 121x329 [12,0]
              CC011: JViewport 118x326 [1,1]
                CC012: JList 118x326 [0,0]
              CC014: JScrollPane$ScrollBar 0x0 [0,0] <INVISIBL
              CC017: JScrollPane$ScrollBar 0x0 [0,0] <INVISIBL
          CC020: JPanel 233x205 [202,12]
        CC035: JPanel 453x413 [2,24] <INVISIBLE>
```

```
CC007: mouseMoved: class java.awt.event.MouseEvent
CC007: mouseMoved: class java.awt.event.MouseEvent
CC007: mouseMoved: class java.awt.event.MouseEvent
CC007: mouseExited: class java.awt.event.MouseEvent
CC006: mouseEntered: class java.awt.event.MouseEvent
CC006: mouseMoved: class java.awt.event.MouseEvent
CC006: mouseMoved: class java.awt.event.MouseEvent
CC006: mouseMoved: class java.awt.event.MouseEvent
CC006: mouseMoved: class java.awt.event.MouseEvent
CC006: mouseMoved: class java.awt.event.MouseEvent
CC006: mouseMoved: class java.awt.event.MouseEvent
CC006: mouseMoved: class java.awt.event.MouseEvent
CC006: mouseMoved: class java.awt.event.MouseEvent
CC006: mouseExited: class java.awt.event.MouseEvent
```

Figure 5. A sample solution in an interactive window on the left and the exploration window showing the widget hierarchy at the top and the events from interaction.

each of the candidate solutions along with an *Accept* and a *Reject* button for each. The solutions are ordered according to their score. In addition, by clicking on the solution itself, the programmer will bring up two windows, one containing the solution that the user can interact with, and a second one that displays a tree showing the widget hierarchy of the solution at the top and a display of all the events that occur when the user interacts with the window at the bottom. An example can be seen in Figure 5. The interaction window provides the user with the option of accepting or rejecting the given solution.

The hierarchy display can be used to do simple editing of the solution in order to let the programmer explore the interface. This currently includes changing labels and making components visible or invisible.

Once the user is done perusing the returned solutions, they can hit the button at the bottom of the panel in Figure 3 to get corresponding code for any accepted solutions. If no solutions are accepted, then the back end will attempt to continue the search to find additional solutions. If solutions are accepted, the code will be returned in a browsable window such as that shown in Figure 4. The user can cut and paste the code from here into their application. Along with the actual code, information is available about the license under which the code is released.

7. EXPERIENCE

To test and evaluate our approach for generating user interfaces using code search, we first obtained sketches of user interfaces. We did this by doing a web search for images using "user interface sketch". We then culled the result for usable sketches that represented potential Java applications (as opposed to web pages or phone applications) and manually converted those sketches into SVG files. In addition to the address book example shown in Figure 1, we used the sketches shown in Figure 6. The test cases then were:

- Login: a sample login screen with a remember me button.
- Pizza: an interface for ordering a pizza with different options.
- Pizza1: similar to Pizza except we only have one list for ingredients rather than two.
- Phone: an interface for making a phone call.
- Mail: a mail reader interface.
- Student: a front end to a student information system.
- Comment: an interface for entering comments in a guest book.
- Address: an interface for maintaining an address book

For each example we tried appropriate keywords. Finding the right set of keywords required some trial and error and we eventually developed a front end for code search that made this easier [38].

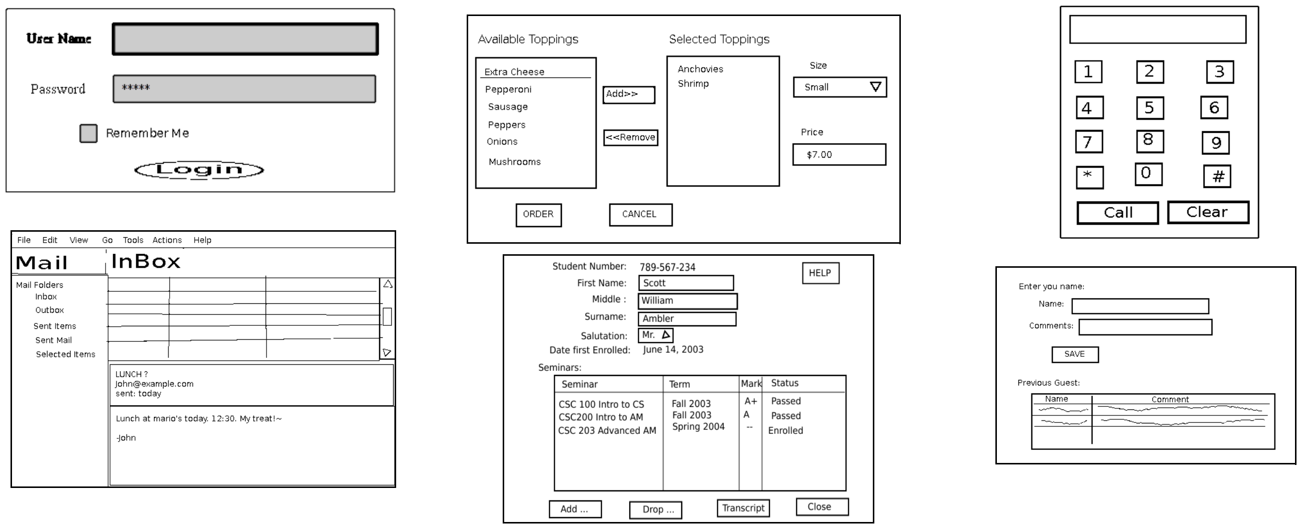


Figure 6. User interface sketches used for testing in addition to the address book of Figure 1. From left to right these are Login, Pizza, Phone, Mail, Student, and Comment.

Table 1: Experimental Results

Test	Keywords	Engine	Scope	Potential Solutions Initial/Total/ Runnable/Tested	Found	Time
Login	login jcheckbox jpassword	OHLOH	FILE	138/2014/453/101	45	3:00
Login	login jcheckbox jpassword	GITHUB	FILE	89/977/231/44	18	1:42
Pizza	jlist jbutton jtextfield jcombobox restaurant	GITHUB+ OHLOH	FILE	15/1284/142/38	17	2:38
Pizza1	jlist jbutton jtextfield jcombobox restaurant	GITHUB	FILE	8/415/20/9	7	0:43
Mail	tree text editor button mail	GITHUB	FILE	88/626/48/15	5	1:02
Mail	tree text editor button mail	GITHUB	PACKAGE	96/46502/351/2	2*	226:33
Mail	tree text editor button mail	GITHUB	SYSTEM	100/47415/340/2	2*	303:27
Phone	phone jbutton	OHLOH	FILE	127/1077/184/9	8	1:15
Phone	phone jbutton	GITHUB	FILE	109/1067/199/14	14	1:23
Student	student jbutton jtext jtable	OHLOH	FILE	91/499/500/47	47*	3:52
Comment	feedback jtextfield jtable	OHLOH	FILE	112/4977/500/178	114*	8:02
Address	name email phone jlist	OHLOH	FILE	133/4122/500/79	131*	4:16
Address	name email phone jlist	GITHUB	FILE	121/3902/500/154	66*	5:48
Address	name email phone jlist	GITHUB	PACKAGE	115/24650/371/25	19*	69:15

We also attempted to find the interface within a file where possible, but did some experiments with larger scopes. For each case we looked at the results that were returned and verified that at least one of the results was a relevant good match for the initial sketch. A summary of the experiments is shown in Table 1.

The first column of the table indicates the test name. The second column shows the keywords used in the test and the third and fourth indicate the search engine and search scope respectively.

The fifth column (*Potential Solutions*) provides some indication of the work done in the search. The first number is the number of starting solutions derived from the returned search results. This is generally the number of unique files found by the search engine. The second number is the total number of solutions that were generated during the search. The third number is the number of solutions that could be compiled and tested while the fourth is the number of solutions that were judged acceptable. The sixth column reports the number of distinct images that were generated and hence the number of distinct passing tests. This is generally not the same as the number of acceptable solutions since there are often solutions that are variants of the same original source and that generate essentially the same user interface. In this case only one is shown to the user initially (although if that is deemed acceptable, the code for all solutions is returned).

The seventh column (*Time*) of the table indicates the wall time (in minutes and seconds) that the search and testing took. The search was run using eight cores of a sixteen core machine. The process is highly parallelizable and these numbers are dependent on the number of threads being used by the search. The time generally does not include wait time in accessing the underlying search engine since we are caching the initial search results in order to lessen the load on the search engines.

The items which are starred in the sixth (*Found*) column indicate that there might be additional solutions, but that S^6 stopped looking because an initial set of solutions were found. S^6 orders

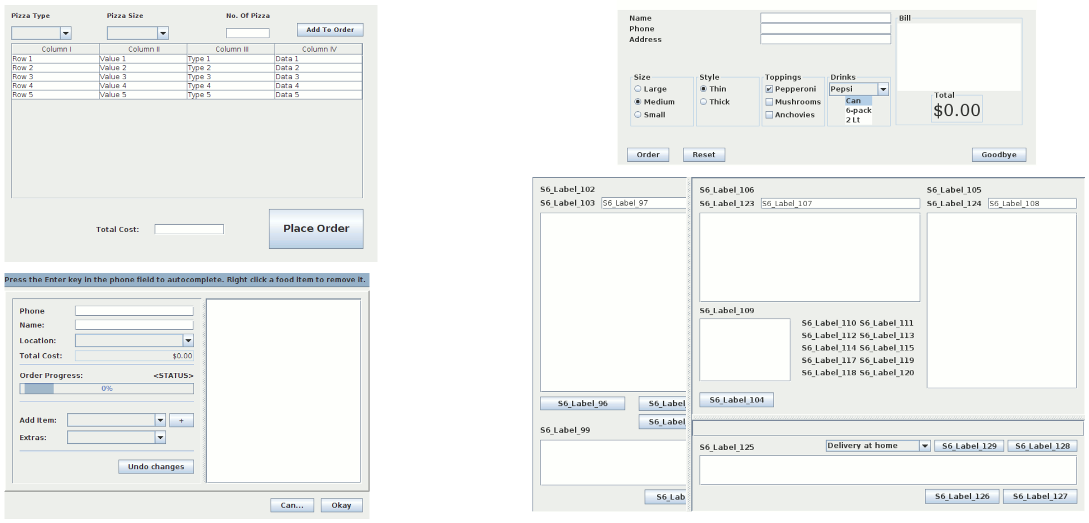


Figure 7. Different Pizza ordering interface found by code search for example Pizza1.

solutions based on their initial ranking from the search engines, the number of transforms used, and a random value, and limits the number of solutions it considers at each stage using this ranking. The unchecked solutions are held in abeyance to be checked later if no other solutions are found. The limits include a maximum of 500 solutions to test and a maximum of 2000 active intermediate solutions.

This can be seen most readily when doing searches at the PACKAGE and SYSTEM levels where there are many more potential solutions to consider. This is the reason that these searches returned fewer solutions than the corresponding FILE searches. These searches take considerable longer. This is due first to the much large set of solutions considered and second to the fact that each of these solutions is substantially larger and hence requires more processing.

7.1 Strengths

While the approach has some problems, it also shows a lot of promise. Using code search it is possible to return working code for a user interface based solely on a sketch and a set of keywords. We see three primary uses for a tool like this.

The first use is to produce an actual working interface similar to the sketch. While this could be done using a user interface builder, there are several advantages to using existing code as a starting point. First, interfaces in the repository have typically been used and tested. Second, such interfaces cover conditions that might not have been anticipated by the original sketch. For example, a sample sketch we did for addresses did not have a zip code field while the generated ones did. Another example is that such interfaces often handle window resizing appropriately, something that today's user interface builders have difficulty with. Third, they typically are more consistent with other interfaces and thus with user expectations. Fourth, they can involve interactivity, both between elements of the sketch and with other pieces of the user interface, for example buttons that are only enabled when fields are completed. Fifth, the code that is returned is often more sophisticated and functional that which would be generated by a user interface builder. In particular, the generated code often included proper layout techniques, more sophisticated hierarchies, scrolling where needed, correct adaptation to changing window sizes, etc. For examples like the phone interface, pushing the buttons entered digits into the display and in some cases the interface allowed typing into the display area; and one of the returned Pizza1 examples automatically updated the prices as items were added to the order. Other interfaces provided validation code that highlighted missing or erroneous fields and that only activated buttons when the inputs needed for them were correct.

A second use for a tool such as we have implemented is to explore a range of user interfaces for an application. User interface sketches, especially those done in the early stages of a project, are usually explorative rather than definitive. Our tool lets user's view and take ideas from other interfaces that are similar or for similar applications. The matching algorithm we use insures that most of the functionality specified by the user is provided, but allows other functionality to be included as well. This is seen in the simplified Pizza interface example with some of the interfaces shown in Figure 7. While these don't match the original sketch, they all are functional interfaces for ordering a pizza, include other components beyond what was initially specified (e.g. thick or thin crust) which the user might have overlooked and want in the final interface.

Central to such exploration is the ability to interact with the user interface to check whether it is appropriate or not. Not only does this show some of the additional features that are included in the code, it also lets the programmer get a better sense of how the interface might be used. For example, one of the address book examples looked at first hand to be unusable because it was a long row of input fields. However, when its shape was changed to be rectangular, it includes all the necessary buttons and functionality — it just uses a simple flow layout rather than a positional one.

A third use for the tool is as a part of a front end for exploring code repositories. In a related project, we developed a tool that uses a programming environment as a front end for searching code repositories [38]. We added a simple interface to this tool that invokes our modified S^6 implementation for a particular piece of code. This lets the user point to a file that implements a user interface and quickly view a diagram of what the interfaces generated by the corresponding code look like.

7.2 Weaknesses

While the approach works in many cases, it still has weaknesses. The first is that the results are very sensitive to the initial selection of keywords. This is due to the fact that the search facilities provided by existing code search engines are primitive by modern standards both because of the search techniques used and the difficulty of mapping keywords to programs. Substantial work has been and continues to be done on code search which should address these problems in the future.

A second problem is the time taken to do the search, especially if the search is at the PACKAGE or SYSTEM scope. The three cases we considered here took hours to complete. There are several difficulties here. First, they tend to yield a large number of potential solutions that need to be explored. Second, the size of these solutions (500k-5M characters), and the complexity of analyzing and transforming solutions of this size, means that each solution takes significantly longer to evaluated. Moreover, the large number of candidates from each solution means that solutions returned early by the initial search tend to dominate and solutions that are returned later are not considered fully or sometimes at all. These are problems that can be addressed by doing a better job of restricting the solutions initially and during the search.

A third problem involves the use of user interface libraries. Our efforts to date have concentrated on code that uses Swing and AWT directly. Complex applications often use a third party user interface library, of which there are quite a few. Code that uses such libraries generally can not be made to compile in a useful manner. It would be relatively simple to extend our tool to let the user indicate which if any third party libraries should be allowed and to incorporate those into the search process.

A fourth problem is that more complex interfaces tend to be tightly integrated with the rest of the application and the rest of the application often depends on external packages or external systems such as databases. For example, mail applications would typically be built using a table model that is tied either to a database of mail messages, to a cache front end, or to a sophisticated imap interface. While we have developed a number of transformations to extract user interface code, additional, more sophisticated transformations would let us find and return more running examples.

A further problem is the quality and nature of the code that is returned. The system returns compilable, running code. The code can be copied and pasted into a user project and used directly. However, the quality of user interface code in the repositories varies widely and some of it should probably not be propagated. Moreover, the code only includes the user interface and hence will need to be modified in order to integrate it into the rest of the system. Many programmers would prefer that code they will have to work on be written in the style and with the conventions they are used to. While some of this can be taken into account by our search tools (they are able to reformat code in standard styles using [35]) or by the Eclipse or similar formatting commands, others should probably be done by code transformation applied to accepted solutions at the request of the user, or by regenerating the code completely from the resultant interfaces.

7.3 Threats To Validity

In addition to the weaknesses cited, there are several things we should note that might affect the utility and efficacy of the approach and the results of the study. These include:

- The set of sketches chosen might not be representative of what programmers are actually interested in. Sketches available on the web tend to be for sample applications, not for real world code.
- The results are dependent on the set of keywords chosen and it is not clear that other users would be able to choose the proper keywords to get appropriate results for a search.
- Our results only look at Java programs with Swing/AWT interfaces. For other languages and user interface libraries the repositories might not have enough samples to let the system find runnable code. Moreover, different and additional transformations would be needed in these cases.
- We have shown that we can return interfaces from code repositories, not that the interfaces that are returned are actually useful, either as starting points or as runnable code. This would require a very different and much more extensive study and is more appropriate after the tool has been further developed.

8. CONCLUSIONS

Our work demonstrates that it is possible to generate a complete, working, interactive user interface from a sketch using code search. We have developed the techniques needed to translate a sketch into something checkable, to extract the proper code from existing code search engines, to transform that code into a program that compiles and runs and includes only the user interface, and then to let the user interact with and select the results of interest.

Some of the lessons learned in the process that will be applicable to future work in this area include:

- With existing code search engines, the results returned are very dependent on the selection of search terms.
- The performance of these techniques is acceptable for interfaces that are contained in a single file; where the necessary code is spread across multiple files, improved performance will be needed.
- Additional transforms would yield additional solutions.
- More flexibility in matching the specifications to the generated user interface lets the technique be used for exploration.
- Similarly, it is often better to have the user under specify the interface, both for finding solutions and to facilitate exploration.
- The code returned is interactive and generally does more than the code that would be generated by a simple user interface builder.
- Additional work is required to transform the resultant code into something that programs would feel comfortable including directly into their applications.

We view this work as a first step in a system that would produce practical, complete working interfaces that programmers could actually use. Further work in this area should include:

- Providing a front end that would translate an arbitrary sketch into a usable SVG diagram.
- Additional transformation to handle more cases. Of particular interest are transformations that separate the user interface from the existing code in order to better handle complex cases and speed up package and system search.
- Extending the techniques to other user interface libraries such as those for Android phones.
- Better code search front ends to get around the sensitivity of code search to the specific engine and the initial keywords.
- Providing additional facilities to let the user edit the resultant display and then having those edits actually change the code.
- Extending the techniques to handle direct manipulation interfaces where the program does actual drawing (not just widgets) and the drawing is based on internal objects.
- Developing the transformations and other techniques needed to return code that programmers would be comfortable in working with and modifying.

The code for our implementation of user interface generation by code search is available as part of the S^6 code search tool and can be found at ftp://ftp.cs.brown.edu/u/spr/s6.tar.gz. The test cases (SVG files) are available upon request.

9. ACKNOWLEDGMENTS

This work is supported by the National Science Foundation grant CCF1130822. Additional support has come from Microsoft and Google.

10. REFERENCES

[1] Marat Akhin, Nikolai Tillmann, Manual Fahndrich, Jonathan de Halleux, and Michal Moskal, "Search by example in touch develop: code search made easy," *Proceedings SUITE 2013*, pp. 5-8 (June 2012).

[2] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes, "Sourcerer: a search engine for open source code supporting structure-based search," *Proceedings ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications 2006*, pp. 682-682 (October 2006).

[3] Andrew Begel, "Codifier: a programmer-centric search user interface," *Workshop on Human-Coputer Interaction and Information Retrieval*, (October 2007).

[4] Bojana Bislimovska, Alessandro Bozzon, Marco Brambilla, and Piero Fraternali, "Search upon UML repositories with text matching techniques," *Proceedings SUITE 2013*, pp. 9-12 (June 2012).

[5]. Wing-Kwan Chan, Hong Cheng, and David Lo, "Searching connected API subgraph via text phrases," pp. 1-11 in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, (2012).

[6] Shih-Chien Chou, Jen-Yen Chen, and Chyan-Goei Chung, "A behavior-based classification and retrieval technique for object-oriented specification reuse," *Software Practice and Experience* **26**(7) pp. 815-832 (July 1996).

[7] Shih-Chien Chou and Yuan-Chien Chen, "Retrieving reusable components with variation points from software product lines," *Information Processing Letters* **99** pp. 106-110 (2006).

[8] Christopher G. Drummond, Dan Ionescu, and Robert C. Holte, "A learning agent that assists the browsing of software libraries," *IEEE Transactions on Software Engineering* **26**(12) pp. 1179-1196 (December 2000).

[9] Michael D. Ernst, Raimondas Lencevisius, and Jeff H. Perkins, "Detection of web service substitutability and composability," *WS-MaTe 2006: International Workshop on Web Services -- Modeling and Testing*, pp. 123-135 (June 2006).

[10] William B. Frakes and Thomas P. Pole, "An empiracal study of representation methods for reusable software components," *IEEE Transactions on Software Engineering* ***20***(8) pp. 617-630 (August 1994).

[11] Krzysztof Z. Gajos, Daniel S. Weld, and Jacob O. Wobbrock, "Decision-theoretic user interface generation," *In Proceedings of the 22nd AAAI Conf. on Artificial Intelligence (AAAI-08}*, (2008).

[12] Ben Goodger, Ian Hickson, David Hyatt, and Chris Waterson, "XML user interface language (XUL) 1.0 Specficiation," http://www.mozilla.org/projects/xul/xul.html (2003).

[13] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby, "A search engine for finding highly relevant applications," *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, (May 2010).

[14] Saul Greenberg, "Toolkits and interface creativity," *Journal on Multimedia Tools and Applications* **32** pp. 139-159 (2007).

[15] Robert J. Hall, "Generalized behavior-based retrieval," *Proceedings International Conference on Software Engineering,93*, pp. 371-380 (May 1993).

[16] Bjorn Hartmann, Leith Abdulla, Manas Mittal, and Scott R. Klemmer, "Authoring sensor based interactions through direct manipulation and pattern matching," *Proceedings of chi 2007: ACM Conference on Human Factors in Computing Systems*, pp. 145-154 (2007).

[17] David Hemer and Peter Lindsay, "Supporting component-based reuse in CARE," *Australian Computer Science Communications* **24**(1) pp. 95-104 (2002).

[18] Raphael Hoffmann and James Fogarty, "Assieme: finding and leveraging implicit references in a web search interface for programmers," *Proceedings UIST 2007*, pp. 13-22 (October 2007).

[19] Werner Janjic, Dietmar Stoll, Philipp Bostan, and Colin Atkinson, "Lowering the barrier to reuse through test- driven search," *SUITE,09*, pp. 21-24 (May 2009).

[20] Werner Janjic and Colin Atkinson, "Leveraging software search and reuse with automated software adaptation," *Proceedings SUITE 2013*, pp. 23-26 (June 2012).

[21] Jun-Jang Jeng and Betty H. C. Cheng, "Specification matching for software reuse: a foundation," *Proceedings ACM Symposium on Software Reuse*, pp. 97-105 (April 1995).

[22] Otavio Lemos, Sushil Bajracharya, Joel Ossher, Ricardo Morla, Paulo Masiero, Pierre Baldi, and Cristina Lopes, "CodeGenie: using test-cases to search and reuse source code," *ASE ,07*, pp. 525-526 (November 2007).

[23] Otavio Augusto Lazzarini Lemos, Sushil Bajracharya, Joel Ossher, Paulo Cesar Masiero, and Cristina Lopes, "A test-driven approach to code search and its application to the reuse of auxiliary functionality," *Information and Software Technology* **53**(4) pp. 294-306 (April 2011).

[24] Daniel Lucredio, Antonio Franciso do Prado, and Eduardo Santana de Almeida, "A survey of software components search and retrieval," *Proceedings EUROMICRO,04*, pp. 152-159 (2004).

[25] Yoelle S. Maarek, Daniel M. Berry, and Gail E. Kaiser, "An information retrieval approach for automatically constructing software libraries," *IEEE Transactions on Software Engineering* **17**(8) pp. 800-813 (August 1991).

[26] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu, "Portfolio: finding relevant functions and their usage," *Proceeding of the 33rd International Conference on Software engineering*, (May 2011).

[27] Gerrit Meixner, Fabio Patern, and Jean Vanderdonckt, "Past, present, and future of model-based user interface development," *i-com* **10**(3) pp. 2-11 (2011).

[28] Rym Mili, Ali Mili, and Roland T. Mittermeir, "Storing and retrieving software components: a refinement based system," *IEEE Transactions on Software Engineering* ***23***(7)(July 1997).

[29] Brad Myers, Scott E. Hudson, and Randy Pausch, "Past, present and future of user interface software tools," *ACM Transactions on Computer-Human Interaction* **7**(1) pp. 3-28 (March 2000).

[30] Jeffrey Nichols and Andrew Faulring, "Automatic interface generation and future user interface tools," *ACM CHI 2005 Workshop on The Future of User Interface Design Tools*, (2005).

[31] Stina Nylander, "Semi-automatic generation of device adapted user interfaces," *UIST conference companion*, (October 2005).

[32] Andy Podgurski and Lynn Pierce, "Retrieving reusable software by sampling behavior," *ACM Transactions on Software Engineering and Methodology* **2**(3) pp. 286-303 (July 1993).

[33] David Raneburger, Roman Popp, and Jean Vanderdonckt, "An automated layout approach for model-driven WIMP-UI generation," *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems (EICS ,12)*, pp. 91-100 (2012).

[34] Steven P. Reiss, "A component model for Internet-scale applications," *Proceedings ASE 2005*, pp. 34-43 (November 2005).

[35] Steven P. Reiss, "Automatic code stylizing," *Proceedings ASE ,07*, pp. 74-83 (November 2007).

[36] Steven P. Reiss, "Semantics-based code search," *International Conference on Software Engineering 2009*, pp. 243-253 (May 2009).

[37] Steven P. Reiss, "Specifying what to search for," *Proceedings SUITE 2009*, (May 2009).

[38] Steven P. Reiss, “Browsing software repositories,” *Unpublished manuscript submitted for publication*, (2014).

[39] Eugene J. Rollins and Jeannette M. Wing, “Specifications as search keys for software libraries,” *Proceedings 8th International Conference on Logic Programming*, pp. 173-187 (1991).

[40] Colin Runciman and Ian Toyn, “Retrieving re-usable software components by polymorphic type,” *Proceedings 4th International Conference on Functional Programming Languages and Computer Architecture*, pp. 166-173 (1989).

[41] Paulo Pinheiro da Silva, “User interface declarative models and development environments: a survey,” *Proceeding of the 7th International Conference on Design, Specficiation, and Verification of Interactive Systems*, pp. 207-226 Springer-Verlag, (2000).

[42] Jamie Starke, Chris Luce, and Jonathan Sillito, “Working with search results,” *SUITE,09*, pp. 53-56 (May 2009).

[43] Vijayan Sugumaran and Veda C. Storey, “A semantic-based approach to component retrieval,” *Advances in Information Systems* **34**(3) pp. 8-24 (2003).

[44] Suresh Thummalapenta and Tao Xie, “PARSEWeb: a programmer assistant for reusing open source code on the web,” *Proceedings ASE,07*, pp. 204-213 (November 2007).

[45] Thung, Ferdian, Wang, Shaowei, Lo, David, and Lawall, Julia, “Automatic recommendation of api methods from feature requests,” *Proceedings of Automated Software Engineering (ASE) 2013*, pp. 290-300 (2013).

[46] Taciana A. Vanderlei, Frederico A. Durao, Alexandre C. Martins, Vinicius C. Garcia, Eduardo S. Almeida, and Silvio R. de L. Meira, “A cooperative classification mechanism for search and retrieval software components,” *Proceedings SAC,07*, pp. 866-871 (March 2007).

[47] Haining Yao and Letha Etzkorn, “Towards a semantic- based approach for software reusable component classification and retrieval,” *ACMSE,04*, pp. 110-115 (April 2004).

[48] Yunwen Ye and Gerhard Fischer, “Supporting reuse by delivering task relevant and personalized information,” *Proceedings International Conference on Software Engineering,02*, pp. 513-523 (May 2002).

[49] Yunwen Ye, “Programming with an intelligent agent,” *IEEE Intelligent Systems* **18**(3) pp. 43-47 (May 2003).