Autom Softw Eng (2018) 25:157–193
https://doi.org/10.1007/s10515-017-0216-3

Seeking the user interface

Steven P. Reiss[1] · Yun Miao[2] · Qi Xin[1]

Received: 18 July 2016 / Accepted: 29 March 2017 / Published online: 7 April 2017

Abstract User interface design and coding can be complex and messy. We describe a system that uses code search to simplify and automate the exploration of such code. We start with a simple sketch of the desired interface along with a set of keywords describing the application context. If necessary, we convert the sketch into a scalable vector graphics diagram. We then use existing code search engines to find results based on the keywords. We look for potential Java-based graphical 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 is useful for exploring alternative interfaces to the initial and for looking at graphical user interfaces in a code repository.

Keywords User interface generation · Code search · Sketching

✉ Steven P. Reiss
spr@cs.brown.edu

Yun Miao
yunmiao@google.com

Qi Xin
qx5@cs.brown.edu

1 Department of Computer Science, Brown University, Providence, RI 02912, USA

2 Google, 1600 Amphitheater Parkway, Mountain View, CA 94043, USA

1 Introduction

Graphical user interfaces are always a challenge to design and create. Their 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. Comparing alternative interfaces or exploring the design space for a task can be problematic. 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 exploring and building graphical user interfaces by letting the programmer rely on the growing repositories of already developed and tested open source applications. Essentially we want to let the programmer view, interact with, and explore the code of user interfaces in these repositories. 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 existing interfaces that are similar to the programmer's design. Such interfaces can then be used to explore alternative solutions and to help programmers perfect their particular design.

Open source code repositories and systems are growing exponentially. OpenHub now claims over 680,000 repositories with over 35 billion lines of code. (This has doubled in the past 3 years.) GitHub is larger. For many applications, programmers have developed and tested viable user interfaces and the results are in the repositories. A programmer might want to build on one of these existing designs.

Our work lets the programmer start with a sketch of the graphical 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 view and check the result interfaces by interacting with them. The source code for the generated interfaces can then be returned to the programmer if desired.

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 is most useful, however, not for returning actual interfaces, but for exploring 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 service for translating user sketches into structured SVG diagrams that can be more easily understood for user interface search.
– A means for translating SVG-based 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.
– Integration of the user interface search facility into a code search front end.

This paper is an extension of our prior work (Reiss 2014b). It elaborates on and extends this work. It includes a description of how we can convert bitmap-based sketches (such as those actually generated by user interface designers) into appropriate SVG diagrams using computer vision techniques in Sect. 5. It describes how the approach can be extended to Android user interfaces where the user interface is defined both in the code and in a resource file in Sect. 6.1. It describes further applications of the approach for exploring user interfaces in repositories in Sect. 8.3.

2 Overview

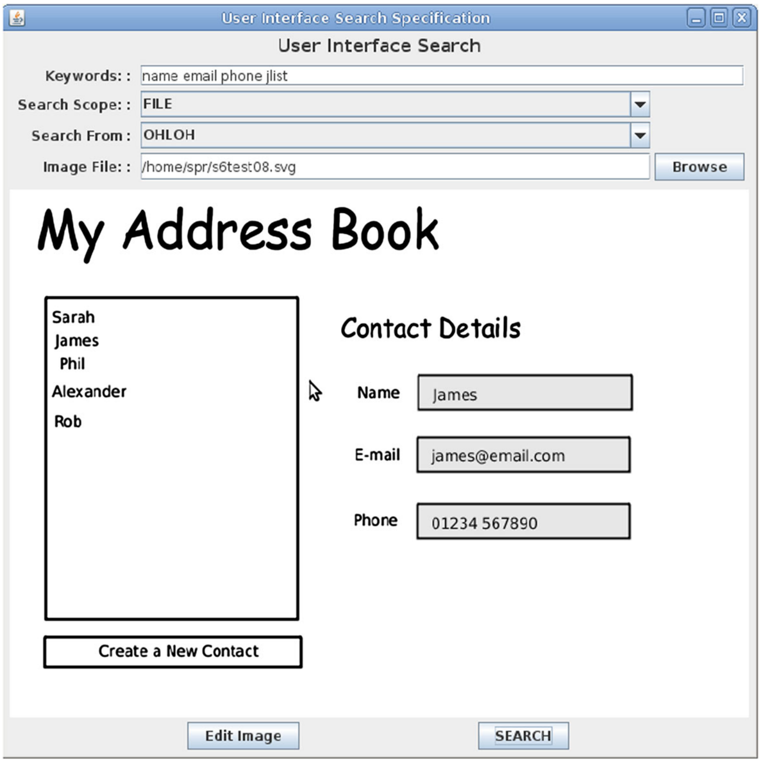
Searching for user interfaces can be broken down into three stages: specifying what to search for, generating candidate solution, and validating those solutions.

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, SUISE, is shown in Fig. 1. Sketching a proposed user interface is typically the first step a user interface designer will take.

The user interface sketch is provided by the user as an SVG file. SVG, or Scalable Vector Graphics, is a standard XML-based vector image format developed by the World Wide Web Consortium and used in web and other applications. If the user has a free-hand sketch of the user interface, we provide a facility to convert that diagram into an appropriate SVG file. This is described in Sect. 5. An SVG diagram is more structured and easier to interpret than an arbitrary sketch. Moreover, 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.

When the user completes the specification (sketch and keywords) and hits the search button, we build a Java code search request for a modified version of our S^6 search engine (Reiss 2009a). (A more complete description of S^6 is given in Sect. 3.3.) To do this we transform the user's sketch into a *hierarchical component description* as described in Sect. 4. 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 or Android widget types that can be used to implement this particular component. The component description from the example shown in Fig. 1 for Swing applications is shown in Fig. 2.

Next, S^6 uses the keywords to find candidate code from an existing code search engine such as Hunter, CodeExplorer, or Github. S^6 next looks for user interfaces in these candidates. A candidate user interface solution, representing a potential user



Keywords: : name email phone jlist
Search Scope: : FILE
Search From : OHLOH
Image File: : /home/spr/s6test08.svg

Name: James
E-mail: james@email.com
Phone: 01234 567890

Fig. 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

interface in the returned code, 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. This process is described in Sect. 6.

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. This is described in the first part of Sect. 7.

The task of exploring user interfaces and deciding which are the most appropriate is left to the programmer. Our tool presents the candidate solutions to the user first by showing thumbnail images of the interface as seen in Fig. 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 as part of a simple application that lets

```
<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>
```

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

the user investigate the interface itself, the widget hierarchy of the interface, and the various events generated by interaction. Once the user has selected a set of acceptable solutions, they can hit the "Get the Code" button to get a display of the resultant code. The code can be used to further explore the interface or to run it again as needed. This is shown in Fig. 4 and explained in Sect. 7.

In the next section we describe S^6 and other related work. Section 8 describes our experiences to date and offers an evaluation of the work.

3 Related work

Creating graphical user interfaces has been a difficult problem since the 1980's when such interfaces started becoming common. 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.

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 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

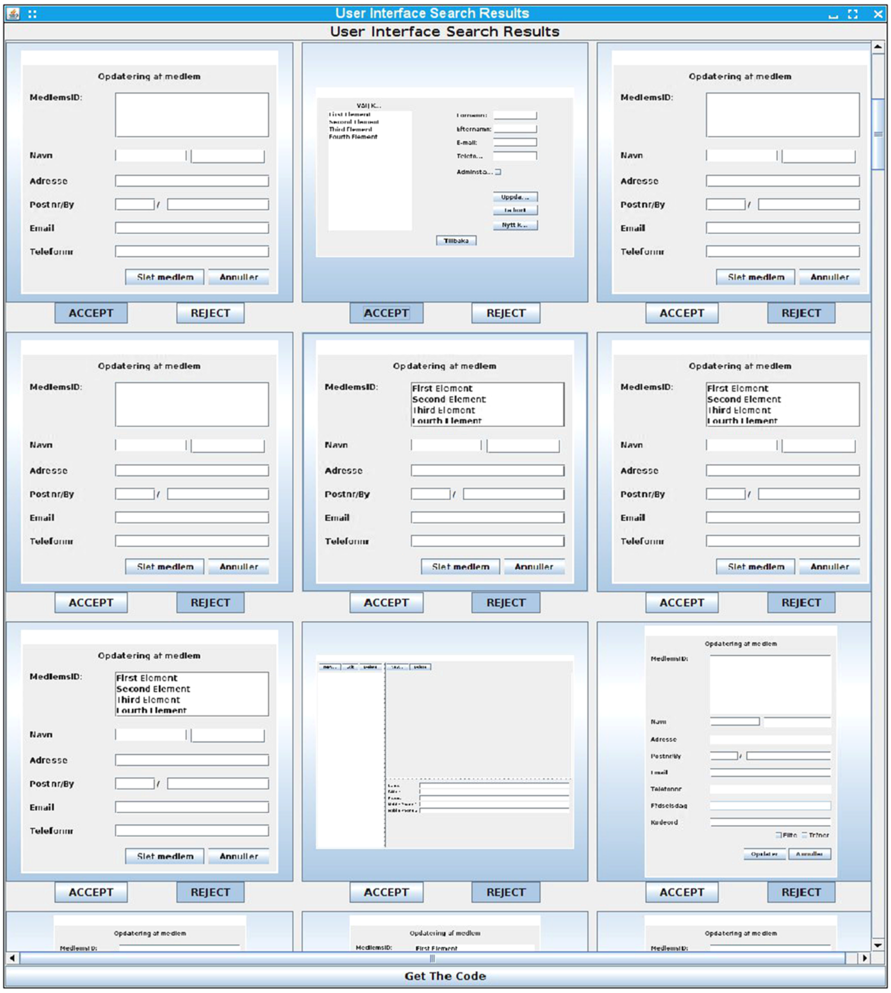
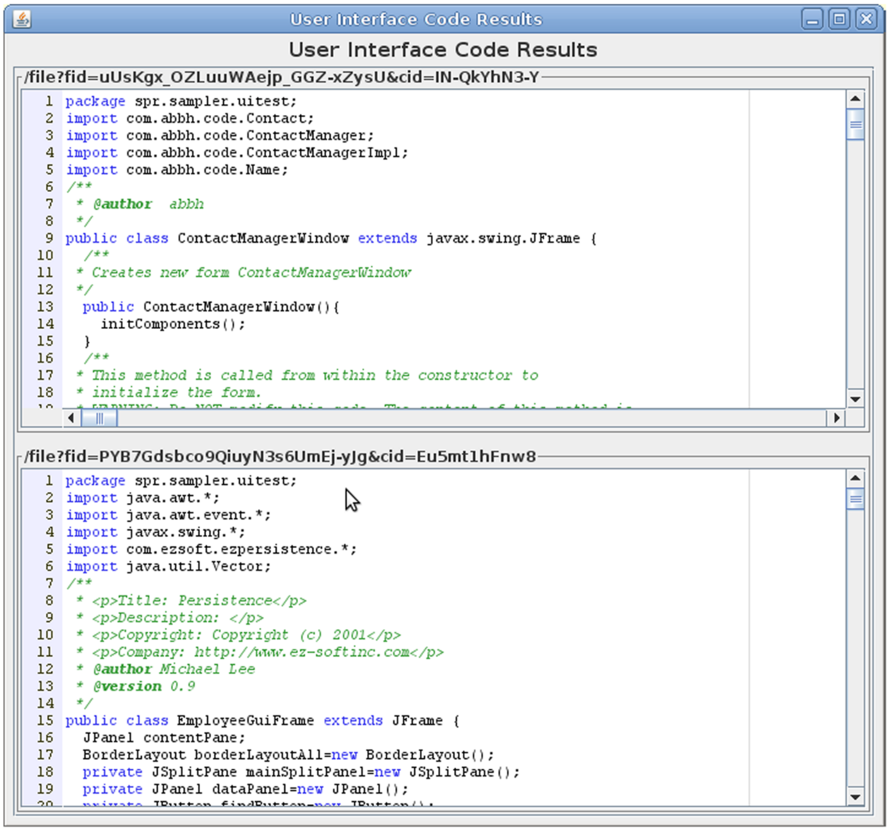


Fig. 3 The resultant display showing thumbnails of potential solutions for the address book sketch of Fig. 1. Each solution can be accepted or rejected by the user. Moreover, the user can experiment with the solution by clicking on it

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 do not handle interaction, data validation, or other interface dynamics; (b) they often use absolute positions and it is complex to generate easily resizable or responsive interfaces; (c) they do not 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,



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

```
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

```
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();
```

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

exploration and comparison of different interface implementations (Hartmann et al. 2007).

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 (Goodger et al. 2003). Others involve developing various models representing the underlying data and the presentation and then generating the interface from these models (Meixner et al. 2011; Raneburger et al. 2012; da Silva 2000). The model driven tools have been more successful when applied to specialized environments (Greenberg 2007; Nichols and Faulring 2005). There has also been work on automatically adapting user interfaces based on device or user constraints (Gajos et al. 2008; Nylander 2005).

There are also tools that let users take sketches as input for generating user interfaces. JavaSketchIt allows creating user interfaces through hand-drawn geometric shape identified through gesture recognition (Caetano et al. 2002). SILK lets designers quickly sketch a graphical user interface using an electronic pad and stylus and then recognizes widgets and other interfaces elements as the designer draws them (Landay and Myers 2001). de Sa et al. (2008) lets users take a picture of a sketch and maps the sketch elements into mobile widgets. More sophisticated sketch-based tools can generate GUI code. MobiDev provides users with a set of predefined elements to draw sketches of applications and then generates the application based on those elements (Seifert et al. 2011). REMAUI infers mobile interface code from screen shots of conceptual drawings using OCR and computer vision techniques but only supports the top three Android widgets (Nguyen and Csallner 2015).

Our work differs in that our focus is on exploring similar interfaces rather than generating the particular interface in the sketch. Where we do generate an interface from existing code, that interface is likely to contain features that are not directly in the sketch, for example the ability to rescale and interactions between different widgets and views.

3.2 Basic code search

There has been significant work done on code search. Early work in this area demonstrated that keywords from comments and variable names were often sufficient for finding reusable routines (Frakes and Pole 1994; Maarek et al. 1991). Later work did query refinement either directly (Sugumaran and Storey 2003), by looking at what the programmer was doing (Ye and Fischer 2002; Ye 2003), using class signatures (Hummel et al. 2008), using an appropriate ontology (Yao and Etzkorn 2004), using the surrounding context (Hill et al. 2009; Wightman et al. 2012), using learning techniques (Drummond et al. 2000), using natural language (Chou and Chen 2006), using an execution trace (Liu et al. 2007), using topic graphs (Wang et al. 2011), using associations (Takuya and Masuhara 2011), using typestates (Mishne et al. 2012), or using collaborative feedback (Vanderlei et al. 2007). Recent approaches, such as Assieme (Hoffmann and Fogarty 2007), Sourcerer (Bajracharya et al. 2006, 2014), Codifier (Begel 2007), Exemplar (Grechanik et al. 2010) and Portfolio (McMillan et al. 2011) expand basic keyword search to consider program structure and semantics. Other recent work has looked at more sophisticated IR techniques (Thomas 2012) and on automatic query reformulation (Haiduc et al. 2012, 2013; Sisman and Kak 2013). More sophisticated search techniques use theorem proving techniques (Stolee and Elbaum 2012; Stolee et al. 2014). These are in addition to commercial tools such as Krugel (krugel.org), Github (www.github.com) and the currently unavailable OpenHub and Google code searches.

While code search shows much promise, it has not caught on extensively. To some extent, this is because the basic code search engines (such as GitHub and OpenHub) are not particularly effective. For example, these search engines are very keyword sensitive, often returning very different results for minor differences in the input. Recent work attempts to address this issue by incorporating textual semantic information (Chan et al. 2012; Lu et al. 2015), program information (Li et al. 2016; Wang et al. 2016; Park et al. 2014), and information from users (Wang et al. 2014; Lin et al. 2014). 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 (Reiss 2009a, b).

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, Google Code Search, Hunter, and Sourcerer (Bajracharya et al. 2006) have been used] 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 simple ones such as changing 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 returns the resultant code to the user. It gives the user the option of different formatting styles (Reiss 2007) 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 that 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

Another semantic approach involves defining the behavior to searched for. This was originally given as input-output pairs (Podgurski and Pierce 1993), and then generalized to allow slightly more flexible specifications (Chou et al. 1996; Hall 1993). More recent work in this area includes PARSEWeb that does static analysis on code fragments found by a text-based code search engine and then looks at input-output types (Thummalapenta and Xie 2007). Other techniques such as program patterns (Paul and Prakash 1994; Wang et al. 2013) and keyword programming (Little and Miller 2007) are designed to work at the level of a code fragment.

Several search-based systems use test cases as input. CodeGenie (Lemos et al. 2007, 2009) lets the user define tests as part of the development process in Eclipse and then uses the method names and signatures from the test to build a query. It uses an internal search engine that understands program structure to find code to test and then presents the result to the user. Other recent code search working on test cases includes (Akhin et al. 2012; Janjic et al. 2009; Lemos et al. 2011) and our S^6. Test cases and semantics have also been used in a similar fashion for finding web services (Ernst et al. 2006; Reiss 2005), but have the problem that the user must know exactly what is being searched for (Janjic and Atkinson 2012).

4 Specifying user interfaces

Our goals in specifying a user interface for code search are three-fold. First, we need to provide an appropriate starting point for the search process. Second, we want to use a natural metaphor, starting with sketches as designers typically do. Third, we want to be able to check the result against the specification so we could test if a generated search solution is 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 (generated by the tool). Based on our own experience and on initial results obtained by browsing the various open source repositories, 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.

For system-based solutions, 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 moves all the files to a single package, updates names and imports statements accordingly.

The remainder of the specification is an SVG sketch of the desired interface. We assume that the user either creates the sketch using an SVG editor or makes use of our SVG conversion facility described in the next section before running SUITE. 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 shown in Fig. 1 will bring up an appropriate editor, either Inkscape if it is installed on the system or GLIPS Graffiti otherwise. 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 (2 pixels or more). Lines are either lines or are boxes that have essentially one dimension (less than 3 pixels). 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 shapes 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 such nestings, we build a hierarchy by finding the innermost nesting for each component. Finally, if there is no unique outermost component, we create one. While we cannot guarantee that a hierarchy in the sketch correspond to an actual hierarchy, this seems to be a good assumption based on looking at a wide variety of user interface sketches both from the web and from our courses. For example, if the sketch superimposed two components to reflect different situations at different times, our approach would not work. However, this does not occur frequently. Most of the sketches we have seen have a limited hierarchy and the hierarchy is either reflecting groupings of buttons or the nesting of text inside text areas or buttons.

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. These checks 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 (more than 100 pixels long and less than 30 pixels high) containing one or more symbols, buttons or text strings. A region with only buttons and text is considered a menu bar, while those with symbols is considered a tool bar.
- 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. The smaller dimension needs to be less than 30 pixels, while the longer dimension has to be greater than 3 times the smaller. 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 (more than 100 pixels long and less than 5 pixels high) 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. For example, the checks involving aspect ratios 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 and Android widgets for each component. This is the only part of the specification stage that is dependent on the user interface being generated for a particular widget set.

The resultant component hierarchy for the input shown in Fig. 1 is shown in Fig. 2. The box on the left containing names 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 *JLabel*s or the outermost *Container*.

Our user interface for specifying what to search for can be seen in Fig. 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 SVG diagrams from user sketches

While SVG diagrams are easy to generate, they are not typically what designers use. As such, we developed a separate tool that takes a bitmap sketch file and generates a suitable SVG diagram. This tool lets the user start with a free-form sketch of the interface they are interested in, convert it into an appropriate SVG diagram, and then use the converted diagram as input into the search process. The converter assumes that the users supply either a jpeg or png image. Other data formats can be easily converted into one of these formats using available tools.

In order to convert from a sketch into a structured diagram, the tool needs to identify the various user interface components in the original sketch. To do this, the converter uses techniques from machine learning and computer vision. The components that are needed for the search process described above are text and shapes such as circles, ellipses, rectangles, triangles, polygons, and tables. We use a gradient based method to find points of interest to shapes and to infer shape information. A variation of the Hough Transform (Duda and Hart 1972) is used to detect tables. A number of simplifying assumptions are made: each hand-drawn shape or table is closed; the boundaries of the shapes and tables have at least the thickness of a ball point pen trace; and there is at most one table in the image.

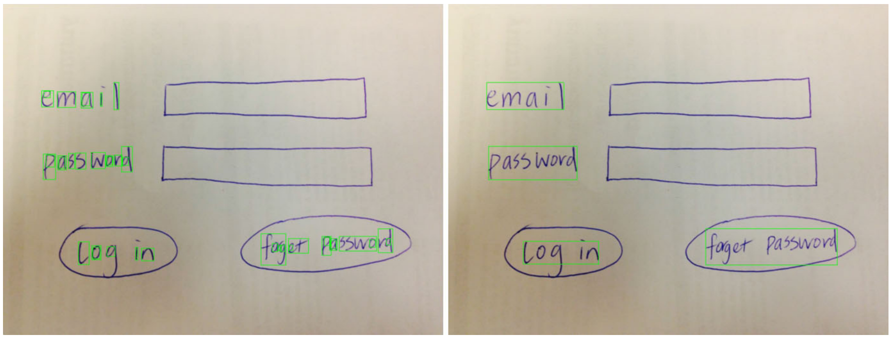


Fig. 5 Text detection results for a sample diagram. The *boxes* on the *left* indicate the individual text contours identified using machine learning. The *boxes* on the *right* show the result after merging positive contours

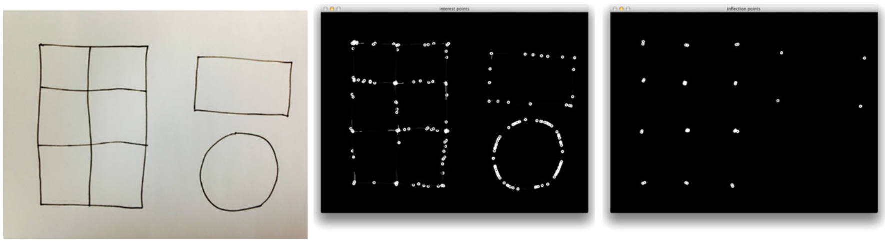


Fig. 6 A shape detection example. The *image* on the *left* shows the original image. The *center image* shows the points of interest detected using gradients. The *image* on the *right* shows the corner points that are detected based on the points of interest

5.1 Text detection

As the first step of our pipeline, we invented a simple contour-based text localization method to help us detect and exclude text regions from future process. A combination of linear support vector machines (SVMs) and adaptive boosting (AdaBoost) is used. The four features we use to train SVMs are: (a) compactness, defined as the ratio of bounding rectangle perimeter over contour perimeter; (b) solidity, defined as the ratio of contour area over convex hull area; (c) contour area; and (d) horizontal crossings (Neumann and Matas 2012). We first convert text into contours and then get a list of text regions by merging adjacent contours based on the classification results. Text regions are then excluded from later shape detection. This is similar to other approaches to text recognition for hand-drawn diagrams (Bull and Gao 2011; Coates et al. 2011; Plamondon and Srihari 2000; Wu et al. 1999). Figure 5 shows sample detection results before and after merging positive contours where the identified contours are shown in green boxes.

5.2 Shape detection

We next developed a novel gradient-based method to detect interesting points in an image. These points are then used to infer the location and types of shapes in the

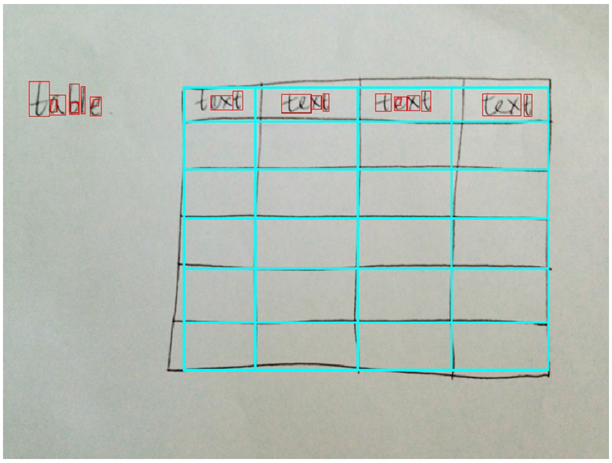


Fig. 7 A sample figure showing what is detected from a figure that contains a table with some text

image. By using gradients, our method is scale-invariant and can detect shapes of all sizes.

We first find inflection points in the image using gradients. Specifically, we compute a gradient vector for each point on a contour, use this to compute the gradient angle change at each point, and choose the local maximum gradient angle change in a two-point window. This effectively filters the contour points by suppressing points with non-maximal local gradient change. This strategy works well because rectangles and the triangles have straight edges. The gradient change between two points on a straight line is close to zero. The gradient is only significantly non-zero at a corner or inflection point. This observation helps us eliminate many unhelpful contour points. The above three steps give us a list of interesting points which are potential corner points on a contour, as seen in Fig. 6.

We next obtain contour points by calculating the gradient angle change between the identified points. This gives us the corner points on the contour. These corner points are then used to determine the shape types and table parameters. Since the input data we have are hand drawings, the edges are not as clearly defined as photos taken by cameras. As a result, local-feature-patch based corner detection algorithm such as Harris corner detection (Harris and Stephens 1988) would work poorly. Toshev, et al. show that the relationship between contour interest points can provide enough information for complex shape detection (Toshev et al. 2012). Since the shape we are trying to detect is a small subset of those in Toshev, et al., we know that we could accomplish our goal by developing a simpler algorithm and extract less data from the contour interest points.

5.3 Table detection

Identifying tables is slightly more complex. A table is usually represented using a set of overlapping rectangles. After the shape information is inferred, we use the list of

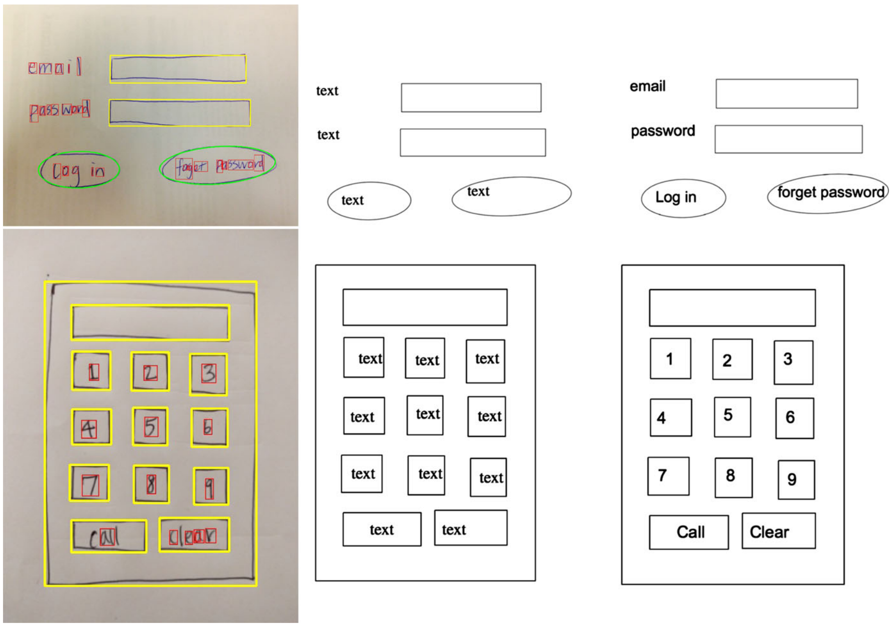


Fig. 8 Examples of the overall process converting a hand-drawn sketch into an appropriate SVG diagram

rectangles to find if there exists a table in the sketch. This is done by finding if there are two or more rectangles that are adjacent either horizontally or vertically. When a table exists, we collect the corner points from all rectangles and use them to determine the position of rows and columns in the table. This is done using a simplified version of the Linear Hough Transform by assuming that our table contains only horizontal and vertical lines. The Linear Hough Transformation is used to detect straight lines in images (Frakes and Pole 1994). Each data point can vote for potential lines that pass through it and the local majority vote determines a line. Normally, a line is determined by two parameters, angle and distance from origin. In our case, we are mostly concerned with vertical and horizontal lines so we reduce the voting space to one parameter. We create 3-pixel-wide voting buckets for each axis. A data point can cast one vote for the x axis, and one vote for the y axis. In the end, the bins with local majority votes will give us the horizontal or vertical lines. An example is shown in Fig. 7.

5.4 Converting the image into SVG

The text and shape detection phases outlined above provide a list of shapes and text regions where each region contains the properties and parameters needed for SVG generation. For example, a line shape contains the two end points, while an ellipse shape contains the center, width, height and the X-axis rotation angle. We generate SVG diagrams using the python library *svgwrite*. Since we did not implement character recognition, we used the general word "text" to represent any text content in the user input. Future work would then include passing the identified text regions to a OCR program to get actual text. Currently, we leave it to the user to insert the proper text using a SVG editor. Figure 8 shows two sample initial images (the top-left and the bottom-left) with the identified shapes marked. The results, with generic texts, are shown in the middle (top and bottom). The top-right and the bottom-right images show the resultant SVG diagrams with proper texts. Based on a set of 30 input images we feed into the shape detector, the following detection result is observed:

Result	Text	Polygon	Table	Circle
Total count	34	61	9	16
False negative	0	4	0	1
False positive	2	3	0	0

6 Generating solutions

Once we have an SVG diagram, we are ready to search for similar interfaces. To do the actual work of searching for a user interface, we modified the S^6 search engine. The modifications generally fall into four categories. The first is handling packages and systems rather than individual functions or classes. These were described in the previous section. The second involves extending the search capabilities to handle Android applications where the user interface is specified in a combination of resource files and code. The third involves restricting the code to that relevant to the user interface by eliminating unnecessary elements. The fourth 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 as a candidate solution.

6.1 Handling android applications

Candidate solutions for Android applications are a bit different since the user interfaces in these applications are generated both from the code and from a set of resource files in a resource directory. Correspondingly, the initial search process we use for Android applications is an extension of the normal S^6 initial search. The Android search process starts with the user-provided keywords and performs two separate searches. The first search looks for Java source files containing the keywords as well as the term "android". Source files are considered here because they are more likely to contain comments that describe the application that might be a good match for the user's keywords. The second search looks for Android manifest files that contain the keywords. This search is restricted to xml files that contain the original keywords as well as "android", “manifest”, “application”, and “activity”. For each of these searches, we look at the first 100 resultant files that are returned from GitHub. Currently, we use GitHub as the search engine since the underlying repository includes not only the source files but also all the related resource files which will be needed to run the program. Other repositories such as OpenHub do not seem to provide access to such files.

We treat each returned file as an indicator of what project should be considered. For each returned file, we search the project for all Android manifest files by doing a project-specific search for xml files using the keywords “android”, “manifest”, “application”, and “activity”. We then restrict the returned set of files to only manifest files and to exclude nested manifest files. Our experience has shown that nested manifests are generally either older versions of the manifest or are not in an appropriate top-level directory with the corresponding resource and source files.

We next process each manifest file that is returned to build an initial solution. We first remove service, receiver, and provider elements of each top level application in the manifest. Then we identify all the classes referenced in the manifest. For each such class *PACKAGE.CLASS*, we do a search using the keywords “package *PACKAGE*” and “class *CLASS*”. We look at the results that are returned and find the one whose path best corresponds to the original manifest. Then we add the source code for this class to the solution for the manifest. Next we add the resource files associated with the manifest to the solution. This is done by walking the directory *res* in the same directory the manifest file is in and finding all relevant files and subdirectories. For each resource that might refer to source files (e.g. xml files in the layout subdirectory), we scan these files for additional class references and, if we find any, we use the above search to find and add the corresponding source files to the solution.

At this point we have constructed a minimal solution that contains a manifest file, all the necessary resource files, and all referenced source files. The system provides three options corresponding to the original file, package, and system-based searches. The first is to use this solution directly. The second is to extend this solution by including all other source files in the same package as one of the referenced files. (This is done by doing a search for “package PACKAGE” for each such package, identifying source files in the package, and adding them to the solution if they are not already there.) The third solution starts with this package solution and does a dependency analysis to find other packages in the project that are referenced by the current source files in the solution. It then adds all source files for these packages to the solution and repeats the process until no new files are added.

6.2 User interface-specific transformations

Once we have found a set of initial solutions, either for Java Swing or for Android, 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*, any non-private method of a class that returns an object that extends *Container*, and top-level widget defined in an Android manifest. 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 or instantiates the Android widget. Where there are multiple possible constructors, we generate separate solutions for each, using logical default values for any parameters on the constructor. If the identified methods take parameters, we generate separate solutions using different default values for those parameters.

Each potential user interface solution is restricted by a transformation that eliminates unnecessary code. This transformation starts with the class that sets up the user interface. It then does a reachability analysis from that class to determine the set of classes, methods, and fields that are required as part of the solution. Any unreachable component is removed from 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 user interface-specific transforms to take these solutions and build new solutions that have a greater possibility of compiling and meeting the user's needs.

The first set of transformations try to modify the code to make it compile while preserving the resultant user interface. These attempt to replace uncompilable code that involves user interface calls with equivalent generic code that will compile.

The first transformation looks at AWT, Swing, or Android method calls and class instance creations (the *new* expressions) for AWT or Swing classes that involve invalid expressions. Invalid expressions can arise from field, method or type references to classes that are not included in the search solution, e.g. the referenced item is outside of the file, package or system or is in non-standard libraries the project uses. The transformation replaces each invalid parameter in the original statements with an appropriate value. Any undefined string parameter is replaced with a unique string; integer and boolean parameters are replaced with a 0 or *false* respectively; color parameters are replaced with a valid color; image and icon parameters are replaced with a known image or icon; parameters that are subtypes of *java.awt.Component* are replaced with a simple label component; and other parameters are replaced with *null*. An example of a call affected by this transformation would be a method that sets the text associated with a widget based on the result of an external call. In this case, the transformation would effectively change the string shown in the user interface to some S^6-specific string but would otherwise preserve the interface.

A second transformation deals with anonymous class definitions that inherit from or implement a Swing, AWT, or Android interface and would not compile because they contain invalid or undefined expressions. This transformation removes the offending internal methods if they are not required as part of an interface or abstract class. Otherwise, it replaces them with methods that simply return default values.

A third transformation looks for calls that use unavailable Swing resources by checking for reference to the *java.util.ResourceBundle* interface. Since the retrieved code in this case does not include resource files or the local resources, calls to standard resource routines will compile correctly but would eventually produce run-time errors. Moreover, these are frequently used in user interfaces to internationalize the code. We replace calls to *ResourceBundle* with calls to an implementation of a resource bundle in our test framework that will return valid values for all requests at run time. We also remove calls to *java.util.Properties.load* which would fail because of the non-existent resource files.

A fourth transformation looks for calls that would cause the user interface to hang or become untestable. For example, if the application attempts to run a modal dialog, the test code, the method returning the component that needs to be checked would never return. Our test code would not be able to access the resultant widget hierarchy. Moreover, the test case will eventually time out and fail, even though a valid interface might be built. Here we remove statements that contain calls to the various *javax.swing.JOptionPane* methods. We also remove calls to *java.awt.Dialog.setVisible* and *System.exit*.

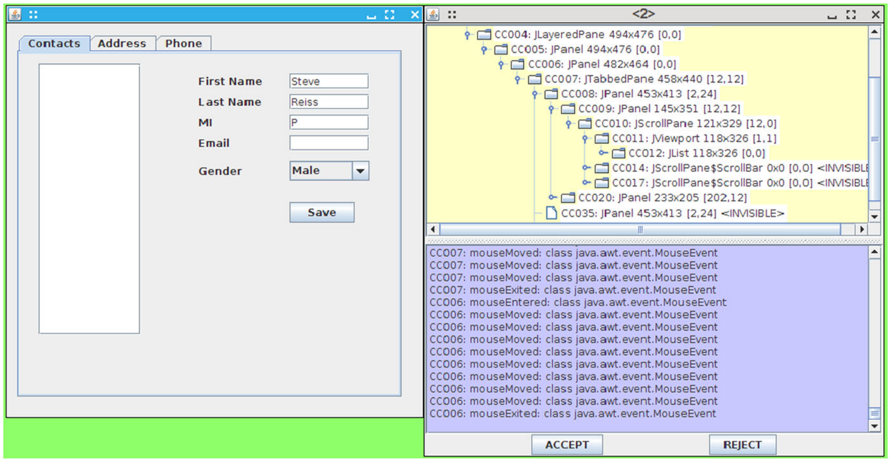
A fifth transformation replaces list, tree, and table models offered by the user with simple internal models provided by our test framework. Many of the compilation and run-time problems that are encountered when trying to run the retrieved code arose because the code attempted to use incomplete or unavailable models. This occurs frequently because such models are themselves integrated with the application and not the user interface. They may be built from external files or from an external database. They might involve classes outside the package or subsystem that was retrieved. Note that this transformation adds additional potential solutions to be considered. It does not remove the original solutions. Thus if the original models would work in the returned code, that solution would still be accepted. The new solutions handle the cases where the original models would not work.

6.3 General transformations

In addition to the interface-specific transformations, there are several generic transforms provided by S^6 that are useful in generating a working user interface from the retrieved code. These generally are designed to clean up the code and ensure it will compile and be testable.

The first relevant transformation handles undefined types or variables. It removes any code that references such variables by removing the corresponding statement when possible. There are cases, however, where this is not possible. For example, if one were to remove a return statement at the end of a method, the result would not compile. In these special cases, the undefined value is replaced with different candidate default values, 0 and 1 (for numerics), *true* and *false* (for booleans), and *null* and non-*null* (for objects). The transformation works repeatedly, so that if a declaration is removed, a later pass will remove all references to the now-undefined variable. The transformation also removes empty statements and private methods that have no remaining statements. Note that the AWT/Swing/Android repair transformations cited above are run before this transformation so that this does not remove any direct user interface code.

Removing code to get it to compile can have various side effects. Additional transformations attempt to handle these. For example, if a variable assignment was removed, the compiler might detect an error in that the variable might be used before it is assigned to. Similarly, removing a statement that might have thrown an exception could make a try-catch block be no longer needed. Additional transformations check for these



```
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$ScrollBar 0x0 [0,0] <INVISIBLE
              CC017: JScrollPane$ScrollBar 0x0 [0,0] <INVISIBLE
          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
```

Fig. 9 A sample solution in an interactive window on the left and the exploration window on the right showing the widget hierarchy at the top and the events from interaction at the bottom

cases and clean up the code. For example, they will add an initial assignment in the declaration of variables that might be used before assigned to, and remove unneeded try-catch blocks.

6.4 Solution filtering

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 some component type that will be needed for each of the non-label user-specified components. Labels are ignored since they are not critical to the resultant user interface. Each element in the hierarchical component specification includes a set of potential widget types. This check ensures that either one of these types corresponds to a simple label, or that a reference to one of the types exists in the code. This reference can either be direct or indirect (i.e. might be to a subclass rather than to the class itself). If there is some non-trivial user component that cannot possibly be implemented by the solution, the solution is discarded. For example, if the user diagram contains a list, a Swing solution would have to contain either component of type *javax.swing.JList*, *java.awt.List*, or *javax.swing.JTable*.

The number of candidate solutions can vary considerably, but generally doesn't become excessive. For example, the search involved with Fig. 1 considered 116 files derived from the OpenHub 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 Fig. 3. The ordering of solutions for testing is a part of S^6 and is a function of the initial rank returned by the search engine, the number of transformations done, and a random value to encourage breadth (Fig. 9).

7 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 tool computes a score describing the quality of the match. The second step is to present the interfaces to the programmer, first by showing a thumbnail 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 a 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. For Java Swing this is an 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. 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 had not 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 user-specified 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 user-specified component needs to match the top-level widget.
- The widget matched with a user-specified 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

```
STRUCT Candidate {
   parent  : Candidate
   spec    : User-Specified Component
   matches : User-Specified Component -> [ Widget x Candidate ]
}

STRUCT Solution {
   matches : User-Specified Component -> Widget
}

Candidate FindCandidates(parent,spec,actual)
   Candidate c = new Candidate(parent,spec)
   FOREACH Direct Child cspec of spec
      cands = FIND all children (recursively) of actual compatible with cspec
      FOREACH cactual IN cands
         nextcand = null
         IF cactual HAS children THEN nestcand = FindCandidates(c,cspec,cactual);
         ADD <cspec->cactual,nestcand> as a potential match in c
      NEXT
   NEXT
   RETURN c

void Match(soln,cand,idx)
   IF too many solutions RETURN
   ELSE IF idx >= # children of cand.spec THEN
      IF cand.parent == null THEN
         USE soln AS a potential solution
      ELSE
         nidx = Index of cand in its parent
         Match(soln,cand.parent,nidx+1)
      END
   ELSE
      child = The idx'th child of cand.spec
      FOR EACH <w,c> in cand.matches(child)
         IF w IS NOT USED in soln THEN
            ADD child -> w TO soln
            IF c != null THEN Match(soln,c,0)
            ELSE Match(soln,cand,idx+1)
            REMOVE child -> w FROM Soln
         END
      END
   END

root = FindCandidates(null,specified_root,actual_root)
solution = new Solution()
Match(solution,root,0)
```

Fig. 10 Solution Validation Algorithm. The function FindCandidates constructs a compact representation of all possible solutions; the function Match finds the actual solutions

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 a maximum number of solutions (currently 1000) have been found.

The algorithm for doing the matching is sketched in Fig. 10. It first uses a recursive method FindCandidates to construct a compact representation of all possible solutions by finding at each level of the hierarchy, the set of all possible widgets that can match any of the child widgets. Note that in finding candidates, we look not only at direct children of the user-specified component, but at all children of the actual widget. Compatibility checking refers to ensuring that the type of the actual widget is either one of the types associated with the user-specified component or is a subtype of one of these. The second phase goes through and enumerates actual solutions. The additional check that is done here is that widgets are not used more than once as part of the solution. This phase stops when a maximum number of solutions are generated.

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 overall score for a given user specified component and a given widget hierarchy is the maximum score computed from any of the accepted solutions.

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 Fig. 3. The programmer is shown the static thumbnail images of 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 Fig. 9. The window on the right provides the user with the option of accepting or rejecting the given solution.

Once the user is done perusing the returned solutions, they can hit the button at the bottom of the panel in Fig. 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 Fig. 4. Along with the actual code, information is available about the license under which the code is released.

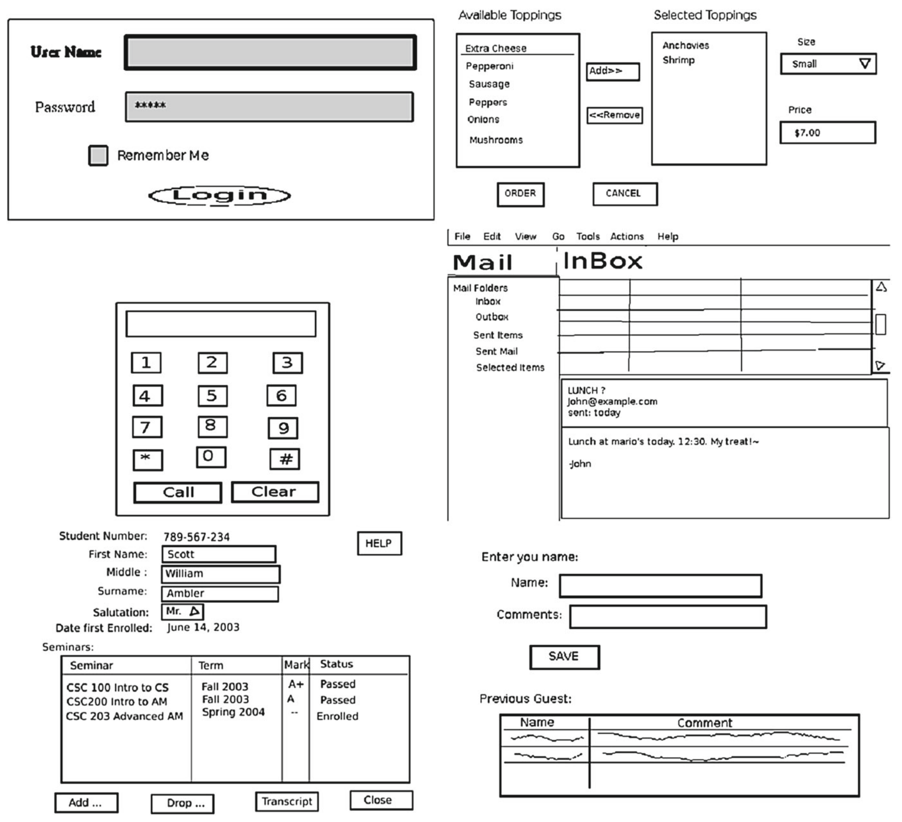
8 Experience

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 represent potential Java applications (as opposed to web pages or phone applications) and manually converted those sketches into SVG files. We later verified that the SVG files were essentially the same (i.e. had the same hierarchy and component types) as those generated by the tool that generated SVG diagrams from user sketches. In addition to the address book example shown in Fig. 1, we used the sketches shown in Fig. 11. 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 (Reiss 2014a). 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. (Note that OHLOH code search was renamed as OpenHub and was recently taken off-line.)

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 keywords basically describe the application and then include the name of one or more of the expected widgets. This type of search worked well, with the descriptive keywords narrowing the search to appropriate applications and the widget keywords eliminating non-Swing applications. Moreover, where we were doing a file-scoped search, requiring both to occur in the same file gave better results.



Extra Cheese
Pepperoni
Sausage
Peppers
Onions
Mushrooms

Anchovies
Shrimp

LUNCH ?
John@example.com
sent: today

Lunch at mario's today. 12:30. My treat!~

-John

Student Number:	789-567-234
First Name:	Scott
Middle :	William
Surname:	Ambler
Salutation:	Mr.
Date first Enrolled:	June 14, 2003

Seminars:

Seminar	Term	Mark	Status
CSC 100 Intro to CS	Fall 2003	A+	Passed
CSC200 Intro to AM	Fall 2003	A	Passed
CSC 203 Advanced AM	Spring 2004	--	Enrolled

Fig. 11 User interface sketches used for testing in addition to the address book of Fig. 1. From *left* to *right*, the *top* two are Login and Pizza, the *middle* two are Phone and Mail, and the *bottom* two are Student and Comment

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 by a user interface programming expert. 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

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

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 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 also 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.

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.

8.1 Generating user interface code

Our original thought was that the tool could be used to produce an actual working graphical user interface that was similar to the user's original sketch, essentially acting as a replacement for the user interface builders that are common in today's development environments. Potentially this would have several advantages. First, user interfaces in the repository have typically been used and tested including user testing in real applications. 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. Third, many interfaces in the repository handle window resizing appropriately, something that many of today's user interface builders have difficulty with. Fourth, interfaces in the repository typically are more consistent with other interfaces and thus with user expectations. Fifth, the generated interfaces can often support 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. For example, in a returned phone interface, pushing the buttons entered digits into the display and allowed typing into the display area; one of the returned Pizza1 examples automatically updated the prices as items were added to the order. Sixth, the code that is returned is often more sophisticated and functional than 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. Moreover, the returned interfaces often provide validation code that highlighted missing or erroneous fields and that only activated buttons when the inputs needed for them were correct.

While generating user interface code might be a worthwhile goal, actually doing so is problematic, and, based on our experience, is not the primary application of our tool. The main issue is the quality and nature of the code that is returned. The system returns compilable, running code that 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 Reiss 2007) or by formatting commands in the user's programming environment, the result may still be code that is not suitable for use in their project. Another issue is that any user interface code that is returned will have to be modified and maintained in the future. This can be difficult to do with external code and much easier if the code is generated by a tool that facilitates such maintenance.

8.2 Browsing potential user interfaces

A more important use for the tool is as a means for exploring user interfaces. User interface sketches, especially those done in the early stages of a project, are usually exploratory rather than definitive. Users often have one or more ideas that they might want to later fill out and complete. The approach taken in SUISE supports this.

First, the user can take a sketch and then view other, existing user interfaces that are similar to the sketch. This lets the user see what might be missing (for example, the zip code field mentioned above), see the range of applications with similar interfaces, and look at alternative layouts and presentations. For example, Fig. 3 shows several different layouts for address book information.

Second, the user can interact with the interface representing the sketch and check whether the interaction is appropriate to the application. This can 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. Interaction can also show how the interface deals with different window or screen sizes, how fields and buttons interact, etc.

Third, the tool can be used as a starting point for exploring a broader range of applications for a potential application. The matching algorithm used by SUISE ensures that most of the functionality specified by the user is provided, but allows other functionality to be included as well. It also allows considerable leeway on the layout and positioning. The user can sketch a simple form of the interface they are interested in and then use the tool to understand a range or alternatives based on that form.

This is seen in the simplified Pizza interface example. Here the sketch included only the minimum that would be needed for a pizza-ordering interface. The interfaces that match this are typically going to be more sophisticated and include additional functionality. Some of the matching interfaces, for example, are shown in Fig. 12. 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 illustrate different approaches that the eventual pizza interface could take.

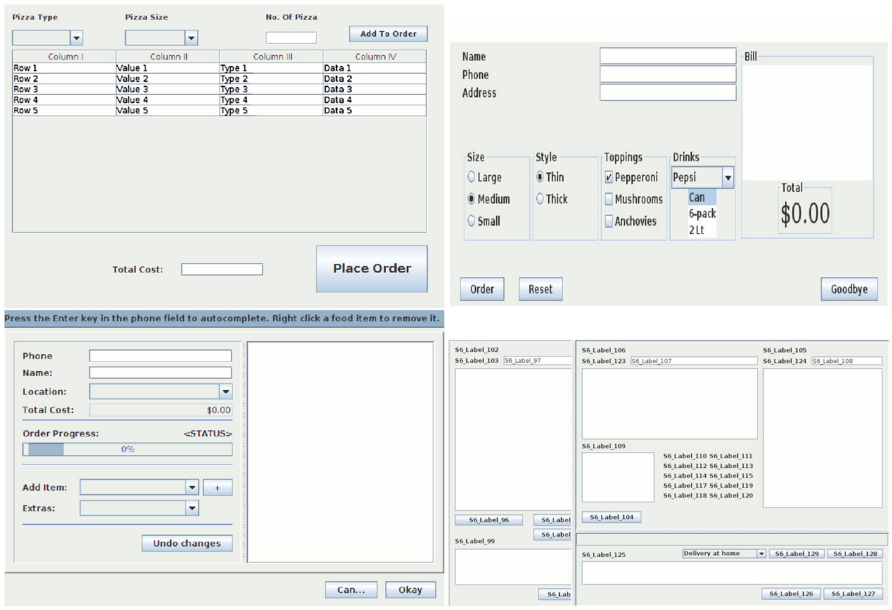


Fig. 12 Different Pizza ordering interfaces found by code search for the Pizza1 example (similar to the Pizza example shown at the top-right of Fig. 6). The *image* on the *left* shows the original image. The *center image* shows the points of interest detected using gradients. The *image* on the *right* shows the corner points that are detected based on the points of interest

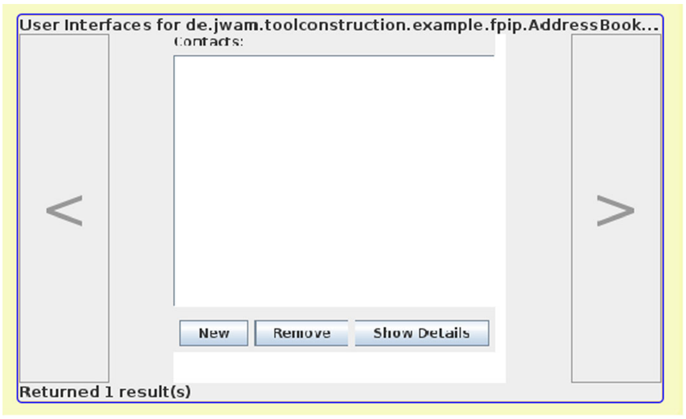

User Interfaces for de.jwam.toolconstruction.example.fpip.AddressBook...
Contacts:
<
>
New
Remove
Show Details
Returned 1 result(s)

Fig. 13 User interface bubble showing the user interfaces inside a repository file

8.3 Browsing user interfaces in a repository

A third use for the tool is as a part of a front end for exploring code repositories. While GitHub and other search engines let the user explore the code in the repository, they do not show the user interfaces that result from that code. The technology included in

SUISE can potentially take a user interface code file and transform it into a running example that the user can either simply view or eventually interact with.

In a related project, we developed a front end for searching code repositories based on the Code Bubbles programming environment (Reiss 2014a). The goal of this project was to provide a better environment whereby the programmer can explore and work with code from existing repositories. We wanted to provide all the navigation features offered by a programming environment such as Code Bubbles in an environment for browsing code repositories. We also wanted to offer an environment that would provide the feedback and editing necessary for the programmer to extract and reuse code from the repository.

Code Bubbles typically runs as a separate tool on top of Eclipse using a message-based plug-in mechanism (Reiss 2012). It includes a small Eclipse plug-in which connects to a message bus that the main environment talks to. Integration is achieved using command messages from Code Bubbles to Eclipse and informational messages from Eclipse to Code Bubbles. Code Bubbles uses Eclipse to handle file access and editing, semantic and syntactic searching, compilation, and execution.

Our approach was to replace the Eclipse plug-in with a repository browsing plug-in. This plug-in handles the same basic set of commands that the Eclipse plug-in does except for those related to debugging. The back end also supports a small set of additional search-oriented commands such as initiating a code search against a set of repositories, augmenting an existing file with other files from same package or project, finding a definition of a name in a given project, returning the next set of results for a code search, marking a package, file, or section of a file as accepted, and outputing all accepted definitions to a given directory.

The back end handles going out to various search engines, extracting the source files, and then simulating the corresponding file hierarchy for Code Bubbles. For each source file, it supports editing and both text and semantic-based search. For each project, it supports compilation on demand (as well as after any edits) with feedback of errors. The back end does its own error-tolerant compilation, necessary because individual files separated from their original package and libraries, will generally not compile cleanly.

Code Bubbles (Bragdon et al. 2010a, b) is an attempt to redesign the user interface to programming, making the programming environment conform to the programmer's working model. It does this by displaying and manipulating complete working sets, collections of task-relevant fragments including code, documentation, test cases, notes, bug reports, and other aspects of programming (Ko et al. 2005; Meyers et al. 2006). The fragments in a working set may be contained in multiple files, classes, or other modules, so quick and easy viewing of all these at once is complicated in traditional IDEs. Code Bubbles presents fragments in fully manipulatable interface elements in order to provide an intuitive arrangement of working sets. The fragments returned from code search, be they methods, classes or files, are a good match for this type of exploratory environment.

Code Bubbles includes a variety of features to simplify and support navigation in addition to what is typically provided by development environments such as Eclipse and NetBeans. It provides a fast search facility that can be used to look at the code hierarchically or to quickly find classes and methods with the given name fragments.

It supports popping up bubbles for definition, references, and implementations by pointing at a name and either hitting a function key or selection off a context menu. It provides class overviews that include comment fragments and can be easily refined. It provides typical environmental facilities to show errors and quickly navigate to them. Our search framework was designed to make use of all these facilities, letting users explore repository code just as they would explore their own projects.

We augmented this interface with a facility to show pictures of the user interfaces that might be implemented by a given file. The user selects the class or file in question and right clicks to select Show User Interfaces. The repository version of Code Bubbles knows the original source of the file or class from the repository and sets up a user interface search request similar to what SUISE would do but specific to that particular source. This request includes a very simple hierarchical component specification consisting of a single empty component. The back end then will access the original code from the repository, apply the various transforms to get it to represent a runnable user interface, and then test it. Any user interfaces that match (regardless of the scores) are then returned to the front end as images. A bubble in the front end then can display the different user interfaces in turn. An example of such a bubble is shown in Fig. 13.

8.4 Weaknesses

While the approach works in many cases, it still has weaknesses. The first is that the results are 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. Code Exchange at UCI, for example, can give very good results initially (Martie et al. 2015; Martie and der Hoek 2013, 2015). We expect that these new approaches will find their way into repository search engines over time and that code search will improve significantly. Since our tool is built on top of existing search engines, we can easily piggy back on these improvements. Moreover, the time our tool takes is small enough when doing file-based search so that a programmer can afford to run it multiple times with different keywords.

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, with some efforts directed toward handling Android applications. 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.

8.5 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. We also assume that the sketches reflect a single graphical user interface, not a combination of interfaces where different items might be superimposed on one another.
– 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 experimental 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.

9 Conclusions

Our work demonstrates that it is possible to find a set of sample graphical user interfaces from existing open source applications that are close to or match an initial designer's sketch for an interface. 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.
- Additional work is required to transform the resultant code into something that programmers would feel comfortable including directly into their applications.

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.

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

References

Akhin, M., Tillmann, N., Fahndrich, M., de Halleux, J., Moskal, M.: Search by example in touch develop: code search made easy. In: Proceedings SUITE 2013, pp. 5–8 (2012)

Bajracharya, S., Ngo, T., Linstead, E., Dou, Y., Rigor, P., Baldi, P., Lopes, C.: Sourcerer: a search engine for open source code supporting structure-based search. In: Proceedings ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications 2006, pp. 682–682 (2006)

Bajracharya, S., Ossher, J., Lopes, C.: Sourcerer: an infrastructure for large-scale collection and analysis of open-source code. Sci. Comput. Program. **79**, 241–259 (2014)

Begel, A.: Codifier: a programmer-centric search user interface. In: Workshop on Human-Computer Interaction and Information Retrieval (2007)

Bragdon, A., Reiss, S.P., Zeleznik, R., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., LaViola Jr. J.J.: Code bubbles: a working set-based interface for code understanding and maintenance. In: Proceedings SIGCHI Conference on Human Factors in Computing Systems, pp. 2503–2512 (2010a)

Bragdon, A., Reiss, S.P., Zeleznik, R., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., LaViola Jr. J.J.: Code bubbles: rethinking the user interface paradigm of integrated development environments. In: ACM/IEEE International Conference on Software Engineering, pp. 455–464 (2010b)

Bull, G., Gao, J.: Classification of hand-written digits using choriograms. In: IEEE International Conference on Image Computing Techniques and Applications (2011)

Caetano, A., Goulart, N., Fonseca, M., Jorge, J.: Javasketchit: issues in sketching the look of user interfaces. In: AAAI Spring Symposium on Sketch Understanding, pp. 9–14 (2002)

Chan, W.-K., Cheng, H., Lo, D.: Searching connected api subgraph via text phrases. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pp. 10:1–10:11. ACM, New York, NY, USA (2012)

Chou, S.-C., Chen, J.-Y., Chung, C.-G.: A behavior-based classification and retrieval technique for object-oriented specification reuse. Softw. Pract. Exp. **26**(7), 815–832 (1996)

Chou, S.-C., Chen, Y.-C.: Retrieving reusable components with variation points from software product lines. Inf. Process. Lett. **99**, 106–110 (2006)

Coates, A., Carpenter, B., Case, C., Satheesh, S., Suresh, B., Wang, T., Wu, D.J., Ng, A.Y.: Text detection and character recognition in scene images with unsupervised feature learning. In: 2011 Internataional Conference on Document Analysis and Recognition ICDAR, pp. 440–445 (2011) da Silva, P.P.: User interface declarative models and development environments: a survey. In: Patern, F. (ed.) Proceeding of the 7th International Conference on Design, Specficiation, and Verification of Interactive Systems, pp. 207–226. Springer, Berlin (2000)

de Sa, M., Carrigo, L., Duarte, L., Reis, T.: A mixed-fidelity prototyping tool for mobile devices. In: Proceedings of the Working Conference on Advanced Visual Interfaces, pp. 225–232 (2008)

Drummond, C.G., Ionescu, D., Holte, R.C.: A learning agent that assists the browsing of software libraries. IEEE Trans. Softw. Eng. **26**(12), 1179–1196 (2000)

Duda, R.O., Hart, P.E.: Use of the hough transformation to detect lines and curves in pictures. Commun. ACM **15**, 11–15 (1972)

Ernst, M.D., Lencevisius, R., Perkins, J.H.: Detection of web service substitutability and composability. In: WS-MaTe 2006: International Workshop on Web Services—Modeling and Testing, pp. 123–135 (2006)

Frakes, W.B., Pole, T.P.: An empirical study of representation methods for reusable software components. IEEE Trans. Softw. Eng. **20**(8), 617–630 (1994)

Gajos, K.Z., Weld, D.S., Wobbrock, J.O.: Decision-theoretic user interface generation. In: Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-08) (2008)

Goodger, B., Hickson, I., Hyatt, D., Waterson, C.: Xml user interface language (xul) 1.0 specficiation. Technical report. http://www.mozilla.org/projects/xul/xul.html (2003)

Grechanik, M., Fu, C., Xie, Q., McMillan, C., Poshyvanyk, D., Cumby, C.: A search engine for finding highly relevant applications. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (2010)

Greenberg, S.: Toolkits and interface creativity. J. Multimed. Tools Appl. **32**, 139–159 (2007)

Haiduc, S., Bavota, G., Oliveto, R., Marcus, A., Lucia, A.D.: Evaluating the specificity of text retrieval queries to support software engineering tasks. In: Proceedings of the 2012 International Conference on Software Engineering, pp. 1273–1276 (2012)

Haiduc, S., Bavota, G., Marcus, A., Oliveto, R., Lucia, A.D., Menzies, T.: Automatic query reformulations for text retrieval in software engineering. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 842–851 (2013)

Hall, R.J.: Generalized behavior-based retrieval. In: Proceedings International Conference on Software Engineering ’93, pp. 371–380 (1993)

Harris, C., Stephens, M.: A combined corner and edge detector. In: Proceedings of Fourth Alvey Vision Conference, pp. 147–151 (1988)

Hartmann, B., Abdulla, L., Mittal, M., Klemmer, S.R.: Authoring sensor based interactions through direct manipulation and pattern matching. In: Proceedings of chi 2007: ACM Conference on Human Factors in Computing Systems, pp. 145–154 (2007)

Hill, E., Pollock, L., Vijay-Shanker, K.: Automatically capturing source code context of NL-queries for software maintenance and reuse. In: International Conference on Software Engineering (2009)

Hoffmann, R., Fogarty, J.: Assieme: finding and leveraging implicit references in a web search interface for programmers. In: Proceedings UIST 2007, pp. 13–22 (2007)

Hummel, O., Janjic, W., Atkinson, C.: Code conjurer: pulling resusable software out of thin air. IEEE Softw. **25**(5), 45–52 (2008)

Janjic, W., Atkinson, C.: Leveraging software search and reuse with automated software adaptation. In: Proceedings SUITE 2013, pp. 23–26 (2012)

Janjic, W., Stoll, D., Bostan, P., Atkinson, C.: Lowering the barrier to reuse through test-driven search. In: SUITE ’09, pp. 21–24 (2009)

Ko, A.J., Aung, H., Myers, B.A.: Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In: Proceedings of the 27th International Conference on Software Engineering, pp. 126–135 (2005)

Landay, J.A., Myers, B.A.: Sketching interfaces: toward more human interface design. Computer **34**(3), 56–64 (2001)

Lemos, O., Bajracharya, S., Ossher, J., Morla, R., Masiero, P., Baldi, P., Lopes, C.: Codegenie: using test-cases to search and reuse source code. In: ASE ’07, pp. 525–526 (2007)

Lemos, O., Bajracharya, S., Ossher, J., Masiero, P., Lopes, C.: Applying test-driven code search to the reuse of auxiliary functionality. In: Proceedings ACM Symposium on Applied Computing, pp. 476–482 (2009)

Lemos, O.A.L., Bajracharya, S., Ossher, J., Masiero, P.C., Lopes, C.: A test-driven approach to code search and its application to the reuse of auxiliary functionality. Inf. Softw. Technol. **53**(4), 294–306 (2011)

Li, X., Wang, Z., Wang, Q., Yan, S., Xie, T., Mei, H.: Relationship-aware code search for javascript frameworks. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pp. 690–701. ACM, New York, NY, USA (2016)

Lin, Z., Zhao, J., Xie, B.: A graph database based crowdsourcing infrastructure for modelling and searching code structure. In: Proceedings of the 6th Asia-Pacific Symposium on Internetware on Internetware, INTERNETWARE 2014, pp. 15–24. ACM, New York, NY, USA (2014)

Little, G., Miller, R.C.: Keyword programming in java. In: Proceedings ASE 2007, pp. 84–93 (2007)

Liu, D., Marcus, A., Poshyvanyk, D., Rajlich, V.: Feature location via information retrieval based filtering of a single scenario execution trace. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, pp. 234–243 (2007)

Lu, M., Sun, X., Wang, S., Lo, D., Duan, Y.: Query expansion via wordnet for effective code search. In: Proceedings of the 2015 IEEE 22nd International Conference of Software Analysis, Evolution and Reengineering, pp. 545–549 (2015)

Maarek, Y.S., Berry, D.M., Kaiser, G.E.: An information retrieval approach for automatically constructing software libraries. IEEE Trans. Softw. Eng. **17**(8), 800–813 (1991)

Martie, L., der Hoek, A.V.: Toward social-technical code search. In: 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), pp. 101–104 (2013)

Martie, L., der Hoek, A.V.: Sameness: an experiment in code search. In: Proceedings of the 12th Working Conference on Mining Software Repositories, pp. 76–87 (2015)

Martie, L., LaToza, T.D., van der Hoek, A.: Codeexchange: supporting reformulation of code queries in context. In: Proceedings of the 30th International Conference on Automated Software Engineering (2015)

McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., Fu, C.: Portfolio: finding relevant functions and their usage. In: Proceeding of the 33rd International Conference on Software engineering. mgpxfport (2011)

Meixner, G., Patern, F., Vanderdonckt, J.: Past, present, and future of model-based user interface development. i-com **10**(3), 2–11 (2011)

Meyers, B.A., Ko, A.J., Coblenz, M.J., Aung, H.H.: An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Trans. Softw. Eng. **32**(12), 971–987 (2006)

Mishne, A., Shoham, S., Yahav, E.: Typestate-based semantic code search over partial programs. SIGPLAN Not. **47**, 997–1016 (2012)

Neumann, L., Matas, J.: Real-time scene text localization and recognition. In: IEEE Conference on Computer Vision and Pattern Recognitioin (2012)

Nguyen, T.A., Csallner, C.: Reverse engineering mobile application user interfaces with remaui. In: Proceedings of Automated Software Engineering, pp. 248–259 (2015)

Nichols, J., Faulring, A.: Automatic interface generation and future user interface tools. In: ACM CHI 2005 Workshop on The Future of User Interface Design Tools (2005)

Nylander, S.: Semi-automatic generation of device adapted user interfaces. In: UIST conference companion (2005)

Park, J.-W., Lee, M.-W., Roh, J.-W., Hwang, S.-W., Kim, S.: Surfacing code in the dark: an instant clone search approach. Knowl. Inf. Syst. **41**(3), 727–759 (2014)

Paul, S., Prakash, A.: A framework for source code search using program patterns. IEEE Trans. Softw. Eng. **20**(6), 463–475 (1994)

Plamondon, R., Srihari, S.N.: Online and off-line handwriting recognition: a comprehensive survey. IEEE Trans. Pattern Anal. Mach. Intell. **22**(1), 63–84 (2000)

Podgurski, A., Pierce, L.: Retrieving reusable software by sampling behavior. ACM Trans. Softw. Eng. Methodol. **2**(3), 286–303 (1993)

Raneburger, D., Popp, R., Vanderdonckt, J.: An automated layout approach for model-driven wimp-ui generation. In: Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '12), pp. 91–100 (2012)

Reiss, S.P.: A component model for internet-scale applications. In: Proceedings ASE 2005, pp. 34–43 (2005)

Reiss, S.P.: Automatic code stylizing. In: Proceedings ASE '07, pp. 74–83 (2007)

Reiss, S.P.: Semantics-based code search. In: International Conference on Software Engineering 2009, pp. 243–253 (2009a)

Reiss, S.P.: Specifying what to search for. In: Proceedings SUITE 2009 (2009b)

Reiss, S.P.: Plugging in and into code bubbles. In: Proceedings Workshop on Developing Tools as Plug-ins 2012, pp. 55–60 (2012)

Reiss, S.P.: Browsing software repositories. http://www.cs.brown.edu/people/spr/rebuspaper.pdf (2014a)

Reiss, S.P.: Seeking the user interface. In: Proceedings of 29th ACM/IEEE International Conference on Automated Software Engineering, pp. 103–114 (2014b)

Seifert, J., Pfleging, B., del Carmen Valderrama Bahamndez, E., Hermes, M., Rukzio, E., Schmidt, A.: Mobidev: a tool for creating apps on mobile phones. In: Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services, pp. 109–112 (2011)

Sisman, B., Kak, A.C.: Assisting code search with automatic query reformulation for bug localization. In: Proceedings of the 10th Working Conference on Mining Software Repositories, pp. 309–318 (2013)

Stolee, K.T., Elbaum, S.: Toward semantic search via smt solver. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, pp. 1–4 (2012)

Stolee, K.T., Elbaum, S., Dobos, D.: Solving the search for source code. ACM Trans. Softw. Eng. Methodol. **23**(3), 1–45 (2014)

Sugumaran, V., Storey, V.C.: A semantic-based approach to component retrieval. Adv. Inf. Syst. **34**(3), 8–24 (2003)

Takuya, W., Masuhara, H.: A spontaneous code recommendation tool based on associative search. In: Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation, pp. 17–20 (2011)

Thomas, S.W.: Mining Unstructured Software Repositories using IR Models. Ph.D. thesis, Queen's University, Canada. Ph.D. Dissertation (2012)

Thummalapenta, S., Xie, T.: Parseweb: a programmer assistant for reusing open source code on the web. In: Proceedings ASE '07, pp. 204–213 (2007)

Toshev, A., Taskar, B., Danillidis, K.: Shape-based object detection via boundary structure segmentation. Int. J. Comput. Vis. **99**(2), 123–146 (2012)

Vanderlei, T.A., Durao, F.A., Martins, A.C., Garcia, V.C., Almeida, E.S., de L. Meira, S.R.: A cooperative classification mechanism for search and retrieval software components. In: Proceedings SAC '07, pp. 866–871 (2007)

Wang, S., Lo, D., Jiang, L.: Code search via topic-enriched dependence graph matching. In: 18th Working Conference on Reverse Engineering, pp. 119–123 (2011)

Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T., Zhang, D.: Mining succinct and high-coverage api usage patterns from source code. In: Proceedings of the 10th Working Conference on Mining Software Repositories (MSR 2013), pp. 319–328 (2013)

Wang, S., Lo, D., Jiang, L.: Active code search: incorporating user feedback to improve code search relevance. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pp. 677–682. ACM, New York, NY, USA (2014)

Wang, S., Lo, D., Jiang, L.: Autoquery: automatic construction of dependency queries for code search. Autom. Softw. Eng. **23**(3), 393–425 (2016)

Wightman, D., Ye, Z., Brandt, J., Vertegaal, R.: Snipmatch: using source code context to enhance snippet retrieval and parameterization. In: Proceedings of the 25th annual ACM symposium on User Interface Software and Technology, pp. 219–228 (2012)

Wu, V., Manmatha, R., Riseman, E.M.: Textfinder: an automatic system to detect and recognize text in images. IEEE Trans. Pattern Anal. Mach. Intell. **22**, 1224–1229 (1999)

Yao, H., Etzkorn, L.: Towards a semantic-based approach for software reusable component classification and retrieval. In: ACMSE '04, pp. 110–115 (2004)

Ye, Y.: Programming with an intelligent agent. IEEE Intell. Syst. **18**(3), 43–47 (2003)

Ye, Y., Fischer, G.: Supporting reuse by delivering task relevant and personalized information. In: Proceedings International Conference on Software Engineering '02, pp. 513–523 (2002)