This document lists open projects related to my research that might be suitable for students at all levels. These projects are listed in no specific order. I would be happy to provide additional information on any of these projects.

Note that while my research involves a number of different topics, much of it focuses on programming environments, particularly Code Bubbles. Students interested in working with me should try out Code Bubbles on their own before coming to me. (If you have an Eclipse-based project, this should be easy. On the Brown CS Linux systems, run the codebb command in /contrib/bin, and point to your Eclipse project. On your own system, you will have to download Code Bubbles and have a writable Eclipse installation. I would be happy to help if you have problems installing or running the system.) Moreover, if you have any suggestions for improvement or missing features, I would appreciate hearing about them and they might form the basis for an additional project.

These projects are at various skill levels. Many involve advanced programming skills. (The Code Bubbles code base is about 400 KLOC; others range from 10-50 KLOC.) Those that do not require advanced programming generally require a design or HCI background and some experience with programming since they involve user interfaces for programming.

Note also that I am open to student-suggested projects and ideas.

**Warm-Up Projects**

These projects are relatively small (1 day to 1 week) that you can use to start learning a code base, demonstrate your abilities, or just explore.

1. **Graphic note bubbles.** Use an existing graphics editor package to provide graphic notes in Code Bubbles. Current note bubbles are html views with a simple text editor.
2. **Add file-based events for SmartSign.** The idea is to provide the user with some control over the display of the sign without having to change the rules. For example, the user might have a command setbusy that touches a file and a corresponding rule that says if the file has been touched in the last k-minutes, to display BUSY.
3. **Fix Automatic Spelling Correction.** Automatic spelling correction in Code Bubbles uses a string edit distance computation to determine when to make fixes and what fixes are the most likely. The current implementation just does insert/delete/replace and gives them each the same score. A better implementation would make case errors and other common typing mistakes less costly than other errors.
4. **Fix BDYN patching.** Bdyn is a Code Bubbles extension that provides a visualization of what threads are doing during a debugger run. It works by instrumenting the code to catch the important (from its perspective) events. However, the patches sometimes seem to confuse the Eclipse debugger so that it becomes impossible to STEP. Find out why this is and fix it.
5. **Bubbles project editor.** The project editor in Eclipse does not always work. In particular, it has problems with dependent projects (and possibly other things). This seems to be due to changes in the way that Eclipse stores project information (what properties, where, how the classpath is used). Determine what Eclipse wants set and how and fix the dialog/actions to accommodate.

6. **Undo Errors.** The Code Bubbles undo mechanism is complex, possibly overly so. It handles undo on a bubble-by-bubble basis where possible. It also handles undo with a selected region. Although it works most of the time, there are some instances where it gets confused and messes up the file. The task here is to determine exactly what causes the problem (i.e. create a duplicable test case that fails), and to possibly fix it.

7. **Large Window Edit Performance.** Code Bubbles sometimes has performance problems when editing becomes noticeably slow. This generally happens when there is a large bubble (i.e. a large method or a bubble displaying the whole file), but it is not obviously predictable. It might be caused by the code to compute the display for each line, the code to handle potential elisions (and its interaction with the back end), slowdowns in Eclipse searches that are assumed to be fast, or some other cause. The task here is to create a repeatable situation where the editing is too slow, and then to determine exactly what is the cause.

8. **Code Bubbles for Node.JS.** Code Bubbles has a JavaScript based implementation designed to work with Node.JS applications. This is only minimally tested and is not widely used. The task here is to try out the interface, and to draw up a list of what one needs to do to make this version practical and usable. This should include a set of bugs to fix and a set of features to implement.

---

**Programming & User Interface Projects**

These projects require significant programming (1 month – 1 year), but the methods and techniques for doing that programming are known, so that there is not a significant research component. Note that, even so, innovative solutions to the problems might be considered research and worth publishing.

1. **Code prediction.** Currently Code Bubbles uses Eclipse’s code completion facility for this, but there has been significant work on prediction at a larger scale (say the next statement(s)). (The most recent is Aroma from Facebook.) Generally, this uses pattern matching or machine learning. Develop a facility based on some of this work and integrate it into Code Bubbles.

2. **Intelligent edit macros.** Simple text editors (say emacs or VI) can include macros, by recording what the user does or by providing a simple programming language. It would be nice to have such a facility in Code Bubbles. A difficulty is how to record or simulate mouse movements and selection. You might also want to be somewhat intelligent. There has been work (from U Washington a few years back) that provided some ideas here.

3. **Better fault localization.** Code Bubbles includes a tool for exploring why a test case fails. This starts by showing the likely locations of the fault. Currently this uses control flow differences between passing and failing test cases. This standard approach is not particularly accurate. A lot more information is available however, especially since Code Bubbles applies Seede (live programming) to the test case and hence has a good approximation of data flow information. It also has access to edit history, versioning, past test cases, etc. These, along with user input, can give a more accurate analysis. There is a lot of similar work to build on.
4. **Multiple Fixes.** The automatic fix facility of Code Bubbles works reasonably well as long as there is a single thing that needs fixing. (It has some quirks you might want to fix in it. It is a little too aggressive at times, especially with syntax errors; and there should be a common priority scheme for all the fixes so that in the occasional case where two different fixes are applicable, it understands this, either to prioritize one over the other or to punt.) However, there are cases where multiple fixes are desirable, for example if the user mistypes an external class name, the system should correct the class name and then add an import statement. While it is impractical to do this all the time, the facility has the hooks to do it in particular cases. The job here is to determine the relevant set of cases based on user experience and to get the facility working.

5. **Pinball.** The pinball machine on the third floor needs to be repaired and converted to use an Arduino and modern circuitry. This talks involves a) building some hardware (6 display panels with LEDs; a driver board to map Arduino inputs and outputs to the array of switches, lights, and solenoids that are available; possibly a sound board), b) fixing the machine (one or more wires are burnt out), and c) writing an Arduino program to play pinball. I have all the necessary parts at this point, as well as some preliminary circuit diagrams and a code outline.

6. **Memory Visualization.** Dymem is our Java memory visualization tool that shows memory ownership. Right now, it is a stand-alone facility. It would be nice to update it (it has some quirks showing potential cycles of ownership in a meaningful way; and can be a little slow to load with large memory), and to incorporate it as an optional view into the Code Bubbles debugger.

7. **Partner-based Communication.** SLACK and similar facilities are the big thing with group software development these days. The interaction facilities in Code Bubbles are primitive (based on 2009 chat interfaces). Develop a new interaction facility that provides slack-like capabilities while taking advantage of the integration into Code Bubbles. The latter should allow the user to share bubbles and working sets; provide screen images or, even better, shared working sets, etc.

8. **Database Model for Live Programming.** Our live programming system in Code Bubbles, Seede, needs to simulate effects on the outside environment in order to execute the same code repeatedly. It includes models for this for the file system and file input and output. What we would like to add is a model for code that accesses a database – i.e. provide a view of the database to the program that is consistent but does not really change the database. (This is sometimes called mocking the database.) You can lookup prior work in this area. We want a database model at least for SQL (java.sql.*), and possibly for MongoDB as well.

9. **Portable SmartSign.** Our smart sign system is complex, involving considerable (generic) code running on Fred4, code running on Valerie, code running in an Arduino, code running on the tablet that serves as the sign, and a variety of configuration and output files. It should be possible to create a mobile implementation of most of this that runs on the user’s phone using only local sensors.

10. **Port Code Bubbles to use IntelliJ.** Code Bubbles includes a package that is an Eclipse plug-in and communicates with the rest of the environment using messages. The message format is fairly well define and we have developed other back ends (for node.js, python, and code search) using it. IntelliJ seems to becoming more popular and more widely used than Eclipse. We want to
provide a backend implementation using the same messages that will let Code Bubbles be used on top of IntelliJ and hence directly with IntelliJ projects.

11. **Annotation Processing.** Annotations added in Java 8 include some unusual capabilities. One of these is the ability to actually rewrite or create new code based on the annotations. (Effectively the annotations provide a meta-language for writing Java programs akin to the C preprocessor and similar systems.) Some of our tools (SEEDE for live programming and FAIT for on-line flow analysis) work by interpreting abstract syntax trees. However, if the abstract syntax tree returned from parsing is not what is actually in the program (due to the preprocessing), then our tools will not work correctly. We want an annotation processor that will create edited abstract syntax trees.

12. **Bubble Rearranger.** When using Code Bubbles for an extended period (hours say), one typically produces a jumble (cascade) or bubbles that might be locally organized, but are definitely not globally organized. Bubbles provides a button to rearrange the bubbles in the display. It should move things around, remove duplicates or unneeded bubbles, and generally create a logical arrangement on the display. We have lots of information about the bubbles, both in terms of usage (when created, when edited, when last focused on, where it was when edited, frequency of use) and in terms of relationships, both implicit (caller/callee, fields for, uses, invokes) and explicit (user click in this bubble to bring up that one; user created a link between these bubbles). Use this information to provide a good layout. Note that this might be user-specific and you might need to know the user’s preferences, either explicitly or through machine learning.

13. **Code Bubbles Look and Feel.** We developed the look and feel of Code Bubbles about 10 years ago. There have been some changes in terms of what good interactive design is since then. Moreover, other environments (Eclipse/IntelliJ) have added new features or facilities. This project involves updating the overall look and feel of the environment to meet current and future standards. This does not need to be a programming project -- it can simply be a full design mockup.

14. **JavaScript version of S6.** S6 does semantic code search, finding and transforming code from GitHub or a similar source so that it passes a set of user test cases. Currently the system only works for Java programs, but all the Java-specific code (parsing, analysis, transformations, and testing) is in one package. The idea is to create a similar package to handle JavaScript so that one can search for JavaScript functions or classes.

15. **Performance Visualizations in Code Bubbles.** Currently Code Bubbles provides a table of performance information that updates as the program runs during debugging. Internally, the environment collects a lot more information, essentially a trie of all the sampled call trees broken down by thread with count and state information. You should create a visualization that shows this information in a way that is most useful to the programmer, possibly with some customization.

16. **Maven Interface for Code Bubbles.** More and more projects distribute themselves today using Maven, which specifies both the sources, the build instructions, and the set of required libraries. There should be a project creation interface in Code Bubbles that takes a directory with a Maven description file and creates a suitable (Eclipse and Code Bubbles) project from it. A further interface would let the programmer view, create, and edit Maven files using a graphical interface in Code Bubbles.
17. **GitHub Interface for Code Bubbles.** While Code Bubbles already includes a GIT front end (that you might want to enhance), it does not provide any smarts specifically for GitHub. This interface could allow project creation from a GitHub URL, access to the bug reporting and comment features of GitHub, automatic creation and update of a GitHub project, etc.

**Research Projects**

These projects involve a research component (experiments, user studies, etc.) and, if done well, are probably suitable for a conference or workshop paper.

1. **FAIT User Interfaces.** FAIT provides an immediate (within seconds) set of errors and warnings related to security. It also can provide a graph for each individual problem detected attempting to explain that problem. (Graphs range in size from 2 nodes to 8,000 nodes.) We need a user interface in Code Bubbles to display the set of errors and to display the explanations. You should do this in conjunction with user studies to understand what is important to display, and what helps the user understand the problem. You should also create a user interface to help the user specify the security conditions that are relevant to their application, both control and data flow based. This should be accompanied by user studies to determine the most appropriate and usable interface. Interfaces include both how and where to apply annotations and how to identify flow-based events as well as a graphical way of defining the appropriate conditions.

2. **FAIT Tuning Interface.** Key to making FAIT practical is to provide some tuning information as to what is important and what is ignorable in the user’s application and the various libraries it uses. FAIT provides some performance information, but you might need more. Such information can be gleamed from the log or you might augment FAIT to provide it. FAIT also has primitive facilities to scan libraries and produce appropriate summary files, but these need to be further tested and augmented to handle a specific set of security concerns. Once you do this, you should create a graphical user interface to help the user create the necessary resource files.

3. **On-Line Flow Analysis for JavaScript.** FAIT as currently implemented works only on Java programs. Since many back ends for web and mobile applications today use other technology such as Node.JS or React.JS, it is desirable to have similar technology for JavaScript applications. This involves writing or adapting an abstract interpreter for JavaScript and integrating it into FAIT and possibly the Node.JS version of Code Bubbles, Nobbles.

4. **Single User Interface for UPOD.** We have developed four different user interfaces for programming our smart sign. Actually, these interfaces are generic and are built from the specifications of the underlying system, with implementations for our sign, OpenHab, and SmartThings. Obviously, four different interfaces is not a practical solution. The research here involves designing and running user studies to determine what features or each of the interfaces are important, how users want to interact with such systems, what is comprehensible, and what debugging an interface program involves, and then taking the information from these studies and using them to design a single user interface. Further user studies should validate your design.
Masters and Ph.D. Level Projects

These projects are more open-ended and less well defined. They involve first getting an understanding of the relevant literature and then developing a plan of attack based on previous work and new ideas. The actual problem to be addressed in these projects might end up somewhat different from the problem I am suggesting.

1. **Bug Fixing without Test Cases.** Current approaches to automatic bug repair require a suite of test cases that effectively define the program semantics. Unfortunately, test cases are a poor representation of semantics and most bug repair methods generate over-fitted solutions, i.e. solutions that pass the test cases but either do not fix the problem or create other, possibly more serious, problems. Is there a way of doing automatic bug repair that does not depend on a very robust test suite, for example, by using simple interactions with the user, which can do a much better job? Prior work has looked at generating new test cases based on a proposed patch and asking the user whether the original or the patched program is correct, and on sorting proposed patches so that the system tries non-over-fitted patches first.

2. **Categorizing Code Search Returns by Functionality.** We have a long-term goal of using code search as a basis for semi-automatic programming, essentially user-guided automatic programming. The first step in this process is to be able to take a set of code search results and group or categorize them according to their functionality. For example, one might search for “embedded web server”, and then want to form groupings of the returned results so that one group contained web servers that only served files, one group contained web servers that used a dispatch table, and one group contained web servers that used callbacks. For each group, one would like to develop a single interface that shows both the commonalities and the options that characterize or distinguish elements of the group. The subsequent goal would be to let the user edit this interface in order to specify more accurately what they want built.

3. **Generating Test Cases from Flow Examples.** FAIT find potential security problems and can produce an explanation graph showing why it thought there was a problem at a given point. However, it is still up to the programmer to first check if this is actually a problem (FAIT generates an over approximation and can easily include false positives), and then to fix the problem. The first step in automating this process is to validate a problem actually exists by generating a valid test case that exhibits the fault. One way of doing this is to use program synthesis and test case generation technology that uses the explanation graphs that FAIT return. You could augment this process with information specific to the particular security problem.

4. **Semi-Automatic Bug Repair.** The goal of this project is to provide an implementation of bug repair that is practical enough to incorporate into a programming environment using a facility such as Eclipse’s Quick Fix. You could try several approaches. One is to speed up automatic bug repair so that it works fast enough to be practical. Another is to provide multiple solutions as alternatives to the user along with enough information to distinguish between them. Another is to specialize according to the problem, for example fixing particular types of security problems. Another is to determine better orderings for perspective fixes.

5. **User Interface Building from Examples.** Many of today’s programming environments include a user interface builder. (Eclipse stands out because it does not.) However, the interfaces that these builders provide are not particularly friendly or usable. It is difficult to create a user interface without knowing intimate details of the underlying package; it is difficult to create
interfaces that resize or scale appropriately; it is difficult to create an interface that depends heavily on data and state in the application. I would like to see a new user interface builder that is based on the user sketching what they want and the builder doing all the work in creating the proper set of widgets, layouts, etc. to create a usable and flexible interface. You could try to do this directly or by data mining existing interfaces. You could use machine to help find interfaces that users would like more, that are easier to use, or that are more compatible with existing frameworks.

Other Projects
These projects do not involve coding (or if they do, minimal amounts), but they do require a detailed knowledge of and experience with Code Bubbles.

1. **Code Bubbles Publicity.** Code Bubbles is relatively mature and stable. I am using it on a wide variety of projects, from small systems up to 500 KLOC. It would be good to have the system more widely used and to get feedback from a broader set of users. This involves several things. The first is publicity. Since I do not do any social media, it is not widely known that Code Bubbles still exists and is available. Having a Facebook page, posting information about it on relevant sites, creating a domain, etc. might help start the process. Second is that it needs to be easier to install and set up. I have tried to make this easier, but one can better document and possibly even automate the current approach. Third, if we are successful in getting additional users, we might want to set up a discussion forum and a bug database. Fourth, if we have enough people interested, we should make it easier to contribute code to the project, making it a real open source project with multiple developers. This might involve making it easier to download and install and moving to GitHub from SourceForge.