Potential Student Projects

Steven P. Reiss

January 18, 2023

This document list 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. This can be done under FastX. 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. For more information see the Code Bubbles web pages.

These projects are at various skill levels. Many involve advanced programming skills. (The Code Bubbles code base is about 400 KLOC; others range from 10-90 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. The difficulty here is developing a user interface for graphics that occupies very little screen space (notes are typically small) and uses only the left (selection) and right (pop-up menus only) mouse buttons.
2. **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.
3. **Global counter**. Create a data type like AtomicLong that supports at least incrementAndGet and works across processes and time. It can be done with different levels of consistency, as long as the value returned is unique and incrementing within a process (thread).

4. **Bluetooth Discovery**. Create a simple program (preferably, but not necessarily, in Java) that lists all Bluetooth LE (4.0) devices that can be discovered by the current machine (MAC address and any other information) along with its RSSI. This should work on Linux, Mac OS/X and, if possible, Windows. The difficult part of this is finding and understanding an appropriate library.

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 include Aroma from Facebook and CodeGenie.) Generally, this uses pattern matching or machine learning. Develop a facility based on some of this work and integrate it into Code Bubbles. Chat-GPT might also be helpful.
2. **Intelligent Sign Cloud Device**. (This is currently under development. Working on this would mean working and extending the existing code base.) The sign outside my office is a one of with systems dedicated to it and it alone. We want to expand this. Our plan is to provide a cloud based IoT device so that anyone can register and have a sign (or multiple ones). Initially signs will just be web pages that are globally accessible; eventually we hope to support a physical device as well (users can always do what I do and have a Chromebook or another tablet running a kiosk app that displays the web page). We want to integrate this into SmartThings and possibly Alexa and Siri as well. Once all this is finished, we want to develop a phone app that provide a simple interface that lets the user control a set of IoT devices using non-trigger, priority-based rules.
3. **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.
4. **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.
5. **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.

6. **Pinball 2**. The pinball machine is almost working. The current code for pinball is written using the Arduino environment on an Arduino Mega. A desirable goal would be to rewrite this code using a real-time OS (freeRTOS), so that scheduling and timing is more predictable. Another extension would be to create the appropriate sound files for this particular game and encode them so that they can be played directly by the processor. You would need an appropriate amplifier. There is also a hardware problem with the connection between the main board and the displays.
7. **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. This should also look into what Eclipse and IntelliJ provide in terms of visualizing memory and offer similar (or better) facilities.
8. **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.
9. **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.
10. **Port Code Bubbles to use IntelliJ**. (An initial implementation of this has been done. However, this is incomplete and not well tested. Work here would mean working with, debugging, and extending the current code base.) 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 project.
11. **Develop a Code Bubbles Plugin for IntelliJ, Eclipse, or VS Code**. The goal here would be to use the IntelliJ or Eclipse framework directly and view Code Bubbles itself as a plugin. This would enable users to just install the plug in to get the effect of code bubbles, not having to run code bubbles separately. It is unclear whether this is possible or practical, but it could be interesting and make Code Bubbles more accessible.
12. **Java Beans Processing for FAIT**. Java Beans uses Java reflection at run time to essentially modify constructors and other methods. For example using the @AutoWired annotation on a field

effectively modifies the constructor to initialize that field to an appropriate value. Our static flow analyzer has difficulty dealing with the. The idea of this project is to enable the flow analyzer to handle such frameworks. This could be done either by rewriting the Bean code inside the flow analyzer as a special case or by coming up with a more generic framework. This could also be done for continuous execution (SEEDE). This can also be applied to the Mockito library.

13. **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. Note this is more difficult than it sounds and probably requires one to use Code Bubbles extensively first to get a good idea of what the desired result should be. Some of this is obviated by new Code Bubbles features that make unused bubbles disappear over time, but the facility would still be useful.
14. **Code Bubbles Look and Feel**. We developed the look and feel of Code Bubbles about 10 years ago. Outside comments said it looked like it was from the 90s. 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. An additional consideration would be how to optimize the display for smaller screens, for example laptop computers.
15. **JavaScript version of S^6**. S^6 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.
16. **Modern front end for S^6**. The web front end for S6 is written in GWT which is no longer being maintained and is not that widely used. The task here is to rewrite the front end using a more modern framework (say VUE/REACT and Node.JS) while improving the user interface.
17. **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. This could also involve putting information on each line of code (mini graph or some other visualization). It might also want to display information about threading and how different threads or clusters of threads are used in performance.

18. **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. A Gradle interface would also be helpful.
19. **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.
20. **Target Creation Interface for Code Bubbles**. One of the things I sometimes do with my git build files is to create jar files. These include runnable binaries, libraries, resources, and plug-ins. In every case one needs to specify the contents of the jar in terms of which class files should be included, what resource files should be included, and what should be in the manifest file. Much of this can be done using the jar task in ant. What I would like to see is a user interface in Bubbles that lets the user define such targets and would then automatically build them on each save (or on demand). A nice option would be to specify a set of starting classes and then have the tool find all other classes on the project class path (including library classes) that were needed to make a running binary. The user should have the option of including all compiled classes or only a subset. Also, the user should have the option of including or excluding the various libraries on the class path.
21. **Template creation interface for Code Bubbles**. Code Bubbles supports the use of templates for creating new class, interfaces, etc. These can be very convenient. However, there is no user interface or documentation for creating or editing them. The task here is to create such an interface that is intuitive and makes it easy for the user to both create new templates, use existing templates from another project (pretty much done), and perform minor edits to the existing templates. The later would be helpful when reusing templates from another project.
22. **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. Initial interfaces already exist for this purpose, but need to be improved or redone.
23. **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 produces the information needed to accomplish this, and we have developed initial interfaces to display this information. These interfaces are a bit clunky and need to be improved. More importantly, interfaces are needed to let the user update the resource files to effect the appropriate tuning.
24. **Automatic Synchronization of UML and Code**. We want to let a user maintain both a UML class diagram and the set of interfaces (or abstract classes) that can be derived from that diagram in such a way that editing either affects the other. By restricting the code to simple interfaces or abstract classes (to allow constructors and static methods), such a mapping should be pretty much 1-1 and the two-directional update should be doable.

25. **Code Bubbles for Node.JS**. Code Bubbles has a JavaScript based implementation designed to work with Node.JS applications. We have a current implementation to use for developing a node application. There are a number of issues (minor syntax problems tend to yield unassociated warnings; search sometimes fails; debugging doesn't work correctly; linked files don't always work; project editing is incomplete; syntax highlighting is sometimes wrong; …) that need to be addressed to make this totally useful (say before giving it to students).
26. **Associating Information with Lines**. Some IDE plugins can best be viewed through the source by displaying a little information with each line of code. For example, performance data could show the cost of each line. In general, this involves a mini-graph or other graphic that can be added to each line (either at the start or the end) as part of the editor display. Develop a general framework for this that works in Code Bubbles and an implementation of at least one use of the framework. Note that this could be done directly as views (swing term) or as an overlay window on top of the editor. (This has been implemented for argument prompting. Other aspects still open.)
27. **ROSE extensions**. ROSE is our tool for helping the developer debug by suggesting potential repairs based on a simple problem description within the debugger. Work needs to be done on the user interface, on making it easier to use, and on incorporating other repair generation techniques so it has a better success rate.
28. **Coworker notifications**. BUBBLES tracks what changes have been made in other versions of a file by other authors and stores that information. It then, when displaying a file, gathers the relevant information and tries to display a bar in the annotation area showing that others have or are working on the given code. This is problematic in several ways. First, the computation of lines seems to be wrong so that it shows the wrong line or set of lines being worked on. Second, the change notice (tooltip) need help (dates, better information that line numbers, possibly reference to commit statement). Third, the means of notification might be improved – might want to only consider more recent changes, etc. The task here is to make this into a working and useful facility.
29. **Debugging in the Main Bubbles Area**. Bubbles currently switches to a separate viewer to do debugging, forcing the developer to jump back and forth. This causes problems in that sometimes the developer will do some code exploration, etc. while debugging and that would not be available later for continued development. We have partially implemented a facility that creates a working set in the main bubble area that is devoted to debugging. This area grows dynamically, etc., and it is possible to move bubbles easily from here to anywhere else in the bubble area. It is unclear if the user interface for this is intuitive and there are a few quirks in how it works. This project would be to investigate what the best user interface is. It might involve fixing this implementation; it might just involve enabling one to move bubbles or bubble groups from the debugging area to the main area.
30. **Better text coding for errors**. Look at Whole Tomato visual assist. It has several features where they use text highlighting to help developers correct mistakes quickly. See if these can or should be integrated into Code Bubbles either as code highlightings or as auto correctors. Note that they would first have to be looked at from a Java perspective.
31. **Create a Bug Repository for Evaluation**. I have several projects that aim to assist the developer while debugging. These can provide suggested repairs, generate a test case, show where problems might be, or explain why a variable has a certain value. I need to have a test suite that

will let these projects be evaluated. The suite should consist of 20-60 program runs. Each run should start with a main program (launch configuration) and should stop at a breakpoint. This can be an exception, assertion failure, or a regular breakpoint. For each run I need to know the sequence of user inputs needed to get to the breakpoint (can be clicks, etc.). I also need to know the symptoms of what is wrong (that the exception shouldn't be thrown, that variable X has the wrong value, etc.). Finally, I need to know the "correct" fix that would make the program work. This can be done using an existing open-source project (preferably one using a GUI), or other existing code that can be made public. Bugs can be old, reported ones, planted ones, or ones that you find.

32. **Rewrite S6 to work with GPT**. Chat-GPT or GPT-3 provide the ability to find possibly appropriate code snippets. S6 provides the ability to ensure these snippets do what the user wants and fit into their particular program. This seems like an ideal marriage or technologies.
33. **Code Indentation**. The current implementation of indentation in Code Bubbles was cribbed from Eclipse a while ago. It is both out-of-date and, especially for Node.JS, somewhat buggy. The current code has been modified (hacked) considerably to make things work. A clean implementation that works for multiple languages (Java, JavaScript, and possibly Python) would be very helpful.

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. Many might be suitable for masters students or as a first step for Ph.D. students.

1. **On-Line Flow Analysis for JavaScript or Dart**. FAIT as currently implement 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.
2. **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. We have a start working on this, but have not done any of the user studies and the implementation is far from complete.
3. **FAIT Problem Interface.** Security problems are program and system dependent. In order to be practical, FAIT needs a means for the programmer to quickly understand and define the safety conditions (both control-based and data-based) that apply to their system. This interface should help the user specify the security conditions that are relevant to their application. These interfaces should also define how the program relates to the conditions, for example defining how and where to apply annotations and how to identify flow-based events. The key here is

that this interface needs to be simple and intuitive so developers would actually use it. We have a first cut, but it doesn't meet these criteria. To do this right, it should involve user studies to determine what works and what doesn't.

4. **Reset Execution of a Method**. Java debuggers can restart execution at the start of a method on user command to facilitate debugging. This is done automatically if the code is edited and saved in modern environments. However, while execution reverts to the start of the method, the values of variables and even the initial parameters are not restored. This means that execution cannot be duplicated correctly. The task here is to develop a very light weight facility to enable the user or the debugger to restore the necessary values so that a second execution would be essentially the same as the first. This might be doable with transactional memory. Another approach would be to use byte code instrumentation upon loading to cache the necessary values so that they could be restored. Some things, for example, maps and collections, might be special cased (e.g., instead of saving whole map, just keep a set of inverse operations). There have been previous attempts at this which might provide a foundation.
5. **Automatic program repair requiring multiple fixes**. Lots of APR techniques have been proposed, but these have all been limited to fixing problems that require a single repair (or at best the same repair made in multiple locations). Using the ROSE technique of recreating the execution with the repair and comparing it to the original execution, it should be possible to develop a scoring function (using ML?) that tells is a single repair is a good starting point for a multiple-fix repair and then trying to find a second repair to build on top of that. This could be applied to ROSE or to any other APR technique.

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. **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. We currently can gather lots of (very noisy) data, but don't really see how to use it effectively. This project is currently underway, but there is enough work left for several dissertations.
2. **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.

3. **Generating Test Cases from a Fault**. When debugging, one sometimes comes to a situation where the program being debugged gets an error (often a NullPointerException) that the programmer needs to correct. Ideally, the problem is obvious and easily corrected. Sometimes, however, the problem can be more subtle or difficult to correct. In this case, it is desirable to have a test case that can reproduce the error without duplicating the entire run. While generating such a test case in general is difficult, since the program is currently stopped at the point of failure and the whole environment is available through the debugger, it should easier to build the actual test case. A command in Code Bubbles (or Eclipse…) that would generate a test case that yields the same error would be the goal here.
4. **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 an 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. This could start with web pages.
5. **User Interface Building without a Design**. While there are existing systems that create user interfaces from a sketch or that let the user build their interface from the set of available components, the idea of this project would be to generate a user interface without any design input. Instead, think of specifying not what the interface would look like, but what functionality it would need to provide. Suppose we had a language for describing inputs, actions, and outputs as well as the states the user interface could be in and when state transitions occur. This should be enough to generate a graphical user interface. This could be done, for example, using code search or even web search (finding appropriate html pages with the right forms and fields). Here one would map the corresponding found page into a similar representation and then compare the language-based representations. It could also use machine learning to determine what are good and bad interfaces and help the developer or direct the system.
6. **Code Search at Scale**. S^6 works fine for methods with proper keywords. However, it does not scale nicely to larger units or to situations where more complex transformations are needed. Research here would start by improving the initial search, possibly by using multiple sets of related keywords, looking at 1000s rather than 100s of results, and then prioritizing what comes back based on some criteria or ML or both. Next one needs to develop better heuristics to guide the set of transformations so that this is not an endless process. This might include interacting with the user. Finally, one needs better testing strategies, possibly by pretesting or doing a little semantic analysis to determine feasibility. It might also involve returning results as

they become available rather than all at once, different means of specifying what should be found, returning partially correct results, etc.

7. **Dynamic Test Cases**. Test cases have a wide variety of uses, for example automatic bug repair, program synthesis, and code search. However, they are difficult to generate automatically as the result is generally not known. The idea is to take higher-level specifications, for example contracts, UML sequence diagrams, or UML statecharts that describe the desired behavior and convert these into test cases. The simplest way to do this is to first find a way of running the code (i.e. anything that executes a normal run). Then one can add dynamic monitoring capabilities to check the conditions, monitor the sequence of calls, or monitor internal states using code patching techniques and determine if the run exhibited or deviated from the desired behavior which would constitute the result of the test.
8. **Self-Correcting Code**. Suppose we could get automatic bug repair to be practical. How could it then be used by a program to fix itself. The key here is that the program needs to understand when something is going wrong and use this information to identify a fix. Assuming we want to work with large, complex systems, then a formal specification is not possible. Moreover, having a large and complete test suite probably isn't going to help either (since then the bug would have been found already). Suppose we use simple contracts to provide semantics. These are incomplete, but can be used to detect errors. The contract violation can be used to do fault localization. The environment of the caller for prior (or future) correct and incorrect runs can be used as a basis for validating any fixes.

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 could also include writing a user's manual better explaining how to use the system.