CodeBuddy: Collaborative Programming IDE

Mike Shafer
California Institute of Technology
Computer Science
Class of 2011
mike.shafer2009@gmail.com

Nathan Watson
California Institute of Technology
Computer Science
Class of 2012
nwatson@caltech.edu

ABSTRACT

We are building an open-source networked system designed and optimized specifically for creating an efficient work atmosphere for relatively small (2 to 20 people) software development groups by offering an easy-to-install, easy-to-use collaborative editing environment.

1. INTRODUCTION

Often, the success of small-scale software development for start-up technology companies is heavily dependent on the speed with which the developers in the company can create working products, and the agility with which the developers can respond to setbacks, design changes, goal changes, and other similar obstacles.

In order to overcome such obstacles as efficiently as possible, communication between developers must be simple, easy, and fast. Different levels of communication are necessary for different types of problems; for example, while a quick chat may be sufficient to clear up small issues in a piece of code, more substantial consultation is generally necessary when deciding major design revisions. However, it is equally important that this variety of communication methods does not hinder the progress of the project (e.g. by taking a long period of time, or by distracting from the focus of the project).

Our application is intended to address this issue by offering chat and messaging tools integrated into a networked, collaborative programming environment. This offers numerous advantages over traditional offline IDEs, in particular due to the efficiency of communication. Aside from in-person consultation, a programmer can use the same environment for programming, communicating ideas to fellow developers, and showing new code for review to peers.

In addition, the program’s design is intended to be as simple to deploy as possible, such that developers can quickly and easily set up the system on existing computers without having to invest in any sort of specialized server hardware. This is particularly important for targeting small start-ups—start-ups generally may not have a large amount of money to work with, and the small number of employees means that any significant setup time results in a loss of overall productivity for the project. In addition, since the program is open source, end users who are proficient with Java can adapt the program to their specific needs as desired.

2. PRIOR WORK

TopCoder's Competition Arena (downloadable from the contest website, http://www.topcoder.com/tc) is a similar product, but with a different focus. As in our system, the interface of the Arena permits users to chat with each other or message each other privately; however, the Arena is used for different purposes. In particular, the program is used to run competitive programming contests in which different users individually write code to solve set problems, while our system focuses on collaborative editing. As a result, the quality of communication tools in our system is of greater importance. Group messaging and chat for different groups involved in the project can also be added. In addition, the interface for code editing should allow multiple simultaneous connections to encourage a more collaborative experience. (Technical issues with file management may prevent our system from allowing multiple write-connections at the same time, but a single write-connection with multiple read-connections, in combination with the chat interface, should suffice in the project prototype for the collaborative atmosphere.)

Google Docs is based on a similar idea. It allows for collaborative editing of many common types of documents and includes a chat function. Our aim is to have our application embody this concept, while providing additional functionality in terms of what documents are supported (basically focusing on a project's codebase), as well as more advanced permission settings. The main documents that will be supported will be code, though text documents, spreadsheets, and presentations can be included. Through the multi-tier group model, we plan to make it easy for an admin to delegate to some set of users read/write access to some documents, while granting other users only read or no access to these documents.

Naturally, since the project also includes an editor, any code editors are also “prior work” to some extent. Obviously, our project extends beyond the task of local code editing, but we can look to commonly used editors (e.g. Notepad++) for ideas when looking for useful functionality to include in the project. One possible feature would be to have plugins for different text editors which provide synchronization, allowing developers to use their favorite editors outside of our application to edit the code.

3. CURRENT PROGRESS

We have decided to develop our application in the Java Programming Language, primarily due to its ubiquity in the public, its cross-platform capabilities, and its relatively strong offerings in user interface design and networking libraries. The client application can be run from any system with version 6 or later of the Java Runtime Environment (JRE), while the server application can be run from any system with version 6 or later of the Java Development Kit (JDK), along with the JavaDB/Derby database package. Given that Java is still a widespread, common technology[1], we feel that this requirement is unlikely to prevent adoption of the system.

We manage updates and revisions to the code using the popular version control system git. Our code base is open source and publicly available via the website github, at the web address http://github.com/q12m/145-codebuddy.

Development of our system is split into two code packages, the server-side application and the client-side application, which will typically be run on separate computers. It is important to note, however, that the client and server can be run on the same computer if desired, in order to prevent the need for a system dedicated to serving when few computers are available.

3.1 Client Application

The client-side application consists of a graphical user interface (GUI) developed primarily using Java's Swing libraries, with some design support from the drag-and-drop GUI development tools in the NetBeans IDE. Upon start-up of the application, the client user is prompted for a username and password, which is sent to the server for authentication. After a successful login, the user will be presented with a split window offering four main parts.

On the top-left, the user is presented with a tree view of the current active project. This gives a view of all available files for editing and viewing. Following typical Java development practices, the system natively allows files to belong to packages. This view enables on-the-fly creation of new projects, packages, and files. A slight modification of the standard Swing library class JTree was required to implement proper leaf and non-leaf relationships between files and packages. From this view, users are able to interact with the server as well, opting to store or retrieve files from the server for their project, and viewing the complete listing of files that the server has to offer.

On the top-right, the file viewer shows the currently active source code file for editing and reviewing. The text editor offers basic editing features, along with several standard features used in other common programs, such as line numbering, automatic indentation, automatic parenthesis and bracket pairing, multi-step undo, and find/replace. The editor allows multiple files to be opened simultaneously, with a tabbed window for switching between active files quickly. Each file has the following components associated with it: a text string denoting the file name, a JTextPane for displaying the file contents and accepting edits from end users, and an internal type (specified as either "leaf" or "non-leaf") to determine its position in the tree view described above.

On the bottom-left, there is a tabbed pane with two views: compiler/tester output and buddy list. The buddy list displays all users in the project, as well as whether each user is online or not, in order to quickly and easily display who may be available for project discussion. The compiler/tester screen shows the results of any compilation attempts for the project, with a listing of compiler errors if any exist, as well as the standard out and standard error streams produced by the program execution.

[1] StatOwl, a web-based analytics company, reports that over 70% of United States Internet users measured in their data have JRE version 6 installed in their web browser as of April 2011, which is sufficient to run our client application.

The compiler/tester output screen shows the results of any project compilation attempt, with a listing of compiler errors if the attempt fails, or success messages if the attempt succeeds. In addition, it shows the program's standard output and standard error streams to monitor the program's progress.

In the buddy list view, a listing of all users associated with server is displayed, showing whether each user is online or not. In future revisions, usernames will be selectable for group/targeted messaging.

On the bottom-right, we have the chat window, where any user currently logged in can write messages and receive feedback from peers. The chat system is fully operational, using a multithreaded scheme to allow for the sending and receiving of messages without disrupting the functioning of the rest of the application.

3.2 Server Application

The server-side application has no graphical interface; in fact, the application currently does not accept input in any form directly from the user. Instead, all functionality of the server is performed through client-server connections.

When the server is started, it first loads up the necessary dependencies in the Oracle JavaDB/Apache Derby database system. After successfully preparing the database for client connections using the Java Database Connectivity (JDBC) service, the server begins listening on a port for client applications attempting to establish a connection (we have arbitrarily selected TCP port 4444 for connections involving user authentication and chat, and TCP port 4445 for connections involving data transfer, though this is trivially configurable in case of problems). When a client connects, the server spawns a thread to handle the connection, and the first action is authentication: the client sends a username/password pair, and the server thread determines whether the pair is valid using a JavaDB containing all valid users. If the authentication succeeds, the server thread transitions to the post-login state, where its primary functions are accepting input chat messages from users and inserting them into the JavaDB, and servicing chat update requests from clients by doing time-dependent JavaDB queries to send out recently received messages to each client. When the user initiates a file request (whether it is for saving a file to the server or loading a file from the server), a separate server thread is spawned for handling the data transmission to prevent large transfers from interrupting the chat features of the program.

3.2.1 Server Security

Since the server in the CodeBuddy system could be any end user's computer and there is often no good way for such a user to ensure complete security of the computer, the authentication system is built to be robust against an attacker attempting to compromise user accounts. To achieve this, no user account passwords are explicitly stored in the user accounts database; instead, the JavaDB backend stores salted hashes computed from the bcrypt algorithm, developed by Provos and Mazieres in 1999 [1]. This function possesses numerous desirable qualities:

(i) It incorporates salting to defend against rainbow table hash attacks, where an attacker uses a table of precomputed hashes for common passwords to find matches in the serverside password hashes;

(ii) It can be made arbitrarily hard to compute in terms of time complexity to defend against brute force attacks by increasing a work factor parameter, unlike alternatives such as MD5 hashing, which is generally extremely quick to compute;

(iii) There has been no known effective cryptanalysis for significantly reducing the work to compute hashes in the algorithm since the function's introduction 12 years ago;

(iv) The function is free from patents and licensing, so it does not impose any legal restriction on our use.

Using the bcrypt algorithm allows the CodeBuddy system to authenticate users without opening the possibility of compromising account passwords if an attacker gains access to the server computer.

4. CHALLENGES

At various points in the course of the project, we have encountered some setbacks and issues halting progress; for documentation, we have listed some of these issues here.

4.1 Managing Connection State

When creating client-server applications, it is necessary to track the state of a particular client's connection (in particular, whether a client is successfully authenticated in the system or not) accurately on both sides of the connection to avoid error. If this is not done, the client-server pair can reach a deadlock; for example, if the client application erroneously progresses past the login screen to the primary view, the server will refuse to allow the client to access or submit entries into chat, since authentication is required; however, the client application will not permit login, since the program effectively "believes" that it has already done so, thus deactivating the login GUI. Some problems with this still linger in our midterm edition of the program; in particular, if a server-client connection is lost for whatever reason without the client application restarting (faulty connection on either of the two systems, server reboot, etc.), the server reverts to pre-authentication state while the client remains stuck in post-authentication, until the end user manually restarts the client application.

4.2 GUI Design

Owing to the varied nature of our system's offering to client users, the amount of time spent on crafting the GUI has been larger than expected. In particular, effectively any new client-side feature must have a new, custom Swing interface for interaction. Since it is often the case that the exact details of implementation are not known until the implementation is complete, we tend to build the GUI on an as-needed basis, rather than attempting to build up a "complete toolbox" of GUI types from the start as we originally planned. The drawback to this is that, each time a new feature is created, any flaws in the feature must be caught while also catching flaws with the feature's new (and thus untested) interface, increasing the complexity of each new module in the code.

4.3 Cross-Platform Compatibility

One of the major goals of the project is to offer a platform for collaborative software development that can operate on any Java-enabled platform. While this is relatively easy to support on the server side due to the simplistic interface nature of the server component, the client code must be carefully constructed to allow for this usage. Many components have to be considered for a usable cross-platform experience.

From the perspective of formatting, the CodeBuddy client interface must be designed for the usage of many different fonts, since different operating systems very frequently use different sets of fonts. (For an example, in one earlier revision of the code, the editor and line numbering components used different fonts of the same size; on one computer which did not have one of the two default fonts, a substitute of a different size was automatically used, causing the line numbers and code to experience a significant offset, rendering the line numbering system unusable.)

In addition, directory structures on different systems (e.g. Linux uses "path/to/my.file" while Windows uses "path\to\my.file"; note the switch between forward and backslashes) must be considered, since the CodeBuddy system uses the directory/folder to manage code packages in Java. Without careful coding, the directory structure would not be managed properly, causing confusing errors to the end user.

5. DESIRED IMPROVEMENTS

The following is a description of various features that could prove to be useful in the project to make the system usable to the largest possible audience. The goals can be naturally categorized into security-related issues, server-side components, and client-side components.

5.1 Security Improvements

While the user account passwords are stored only as salted hashes as described earlier, an attacker with root access to the server can simply bypass the user account system and retrieve private project data directly from the JavaDB backend without the need to compromise any user accounts, since all chat history and project files are stored unencrypted on the server. Unfortunately, securing these files is significantly more difficult than securing passwords. Passwords can be hashed with "one-way functions" (functions without a known efficient inverse), since the password itself is not necessary for the program; in contrast, chat history and files must be recoverable in order to be useful to the end users. Thus, an encryption scheme on the server must be used.

Since the server must be able to encrypt and decrypt files in order to serve them to users, an adversary with root access to the server has complete access to the encryption and decryption algorithms for the files, so a scheme that obscures the key from even the server itself is vital. One solution to this is to maintain a separate global password for file access, but this is undesirable since it requires every user to memorize two distinct passwords (user account and file access). Instead, we would like each user account password to serve as a file access password as well. This problem is tackled by Kiayias et. al. in their 2007 publication on the subject of group encryption [2], in which any one of a set of passwords can be employed as the encryption/decryption key. Using this system, the files and chat history can be encrypted using the unhashed passwords from the user accounts. As a result, having root access to the server would not compromise the data on the server (although it should be noted that it introduces a potential weakest-point-of-failure issue, where only a single user account must be compromised to gain access to the database).

In addition to the serverside encryption, client-server communications are currently unencrypted, so an attacker could easily listen on the network and possibly obtain valuable data intended to be kept secret, including user account passwords, chat messages, and project files. The standard security package in the Java programming language offers a set of secure client-server communications tools using Transport Layer Security (TLS, formerly Secure Sockets Layer or SSL). In combination with the above upgrade to password security, this should make our environment robust against most malicious behavior assuming good security practices from the end users. There is, of course, the remaining potential security vulnerability of an attacker gaining access to the server and then monitoring the functioning of the program itself, which would effectively enable listening to all client-server activity, but it is unlikely that any level of precautionary programming can create an environment free of any spoofing dangers.

For high-security application of the CodeBuddy system, it may be worthwhile to implement and deploy the 2009 scrypt algorithm of Colin Percival [3], since modern computing has taken a strong turn towards parallel computing (especially with the advancements in GPU computation). The scrypt algorithm addresses this issue by not only demanding arbitrarily long computing times for password hashing and key derivation, but also requiring a large amount of memory, hampering the applicability of parallel techniques. However, this would be only a long-term goal in the security suite of the program, since the serverside and connection faults are significantly more serious issues in the current system.

5.2 Server Functionality Improvements

Currently, the server acts as in the capacity of a code dropbox, where end users can deposit code for their peers to read and edit. This can be improved for a more hands-on environment resembling the one offered in the Google Docs web application, in which documents can be read and edited by multiple users in parallel. The advantage to this system is its enhanced level of interactivity—programmers could see their peers' changes as they happen, allowing real-time review and suggestions similar to the frequently used pair programming technique used in the software engineering industry over a remote connection.

Additionally, the serverside application is currently almost completely non-interactive, with only error messages and no administrative tools or on-the-fly configuration. Thus, any TCP port changes, connection resets, database reconfiguration, or similar tasks require a full reboot of the server. While the reboot process is relatively fast and easy, it is still desirable to have some changes permitted without having to disrupt all client-server connections in the process, since a server interruption by default resets all client connections.

Currently, account setup must be done at the code level, instead of having some user interface for automatic account creation. The server should accept some sort of account creation request, with some option for requiring administrator approval for a new account, since a user account unlocks access to all project files.

5.3 Client Functionality Improvements

One feature that could be desirable is a chat history viewer. It can often be useful to refer back to previous lines of correspondence in software development in order to reduce the inefficiency of repeatedly asking peers similar questions, and in some sense, a chat history can serve as an informal documentation of the development process. Chats are already logged on the serverside, but there is no client-facing interface to access them; to read the chat history in the current system, a user must read the log output of the server directly. Ideally, the chat history should be viewable within the client-side application, with some searchable, timestamp-sorted window to review past correspondence.

Private messaging is also a useful feature that can be implemented within the current chat system. This could be applied for cases where a discussion is not of general interest, but should instead be kept between a pair of developers who are directly involved with the subject of the discussion. This promotes a better system of communication between developers, which is a core function of the CodeBuddy system.

For users on lossy connections, the client program does not handle connection failures particularly gracefully, with most connection issues requiring a restart of the program. This is because we use no notion of a session key or any other persistent embodiment of a client-server connection; thus, if the connection itself is disrupted, it must be restarted. In order to handle these issues more elegantly, some session key system should be employed to enable fast reconnects with no need for restarts in order to prevent the workflow for such users from being frequently disrupted.

For program testing, there is no way to interact with the program's input stream at the moment; thus, any program that expects input cannot be tested in the program. Two possible solutions exist to rectify this deficiency: enabling user input through a dialog box when a program accesses the standard input stream, or enabling the user to specify a text file to redirect to standard input in order to allow at least a basic level of interaction. Ideally, both solutions can be added in the future, since some programs demand interaction, while others are best left to automated, pre-written input files for the purposes of streamlining the process of testing code.

5.3.1 Editor Features

The current code editor lacks some features that often can be useful in the course of software development. Since the comfort level of the editor is ultimately the main make-or-break point of the product, we dedicate a special section of the desired improvements to it specifically.

IDEs such as Eclipse and NetBeans offer on-the-fly code compilation, useful for catching bugs and logical gaps as they happen rather than after the code is written, when the bugs may be much harder to find. As a result, the CodeBuddy editor would need to offer comparable functionality to compete with such IDEs. A possible implementation would be a frequent, regularly-timed background compilation attempt by the editor that would report error messages directly as they are found by the compiler.

In addition, IDEs generally offer utilities for code completion and links to official documentation (Javadoc parsing and viewing, for example, in Java development). At a minimum, javadoc parsing for simplistic documentation would be a very valuable asset to development, minimizing the amount of time spent out-of-program looking up external documentation sources.

There is an apparent shift in some circles of software development towards the integration of multiple languages into single projects, since different programming languages often specialize in different types of tasks (at Facebook, for example, there are components written in PHP, C++, Python, and even Erlang). As a result, restricting the system to only Java development, in the long run, will prohibit effective targeting for these communities. Ideally, since there are proprietary programming languages as well as common usage languages that should be supported, the client editor should be adapted to be user-extensible for language support; users should be able to specify editor and compiler configurations in order to add their own support for the languages they demand. This, in addition to default support for the most common languages in modern usage (Java, C++, Python, and the like), will be a crucial step in maximizing the target audience for the CodeBuddy development system.

5.4 Alternative Directions

Rather than continuing in the direction of a completely stand-alone application, it is possible that reusing components from other projects could bring the CodeBuddy system to a reasonable level of development more quickly and with a fuller feature set. For example, integrating the CodeBuddy messaging and backend with an Eclipse plugin would harness the power of a well-established IDE while still adding the chat and networked features as an enhancement to the IDE. This would reduce the amount of time spent developing features to compete with the IDEs that many people have already learned to use effectively, focusing instead on the critical “social coding” components instead.

6. REFERENCES

[1] Mazieres, D. and Provos, N. June 1999. A Future-Adaptable Password Scheme. USENIX 1999. http://cvs.openbsd.org/papers/bcrypt-paper.pdf.

[2] Kiayias, A., Tsiounis, Y., and Yung, M. January 2007. Group Encryption.

[3] Percival, C. May 2009. Stronger Key Derivation via Sequential Memory-Hard Functions. BSDCan 2009. http://www.tarsnap.com/scrypt/scrypt.pdf.

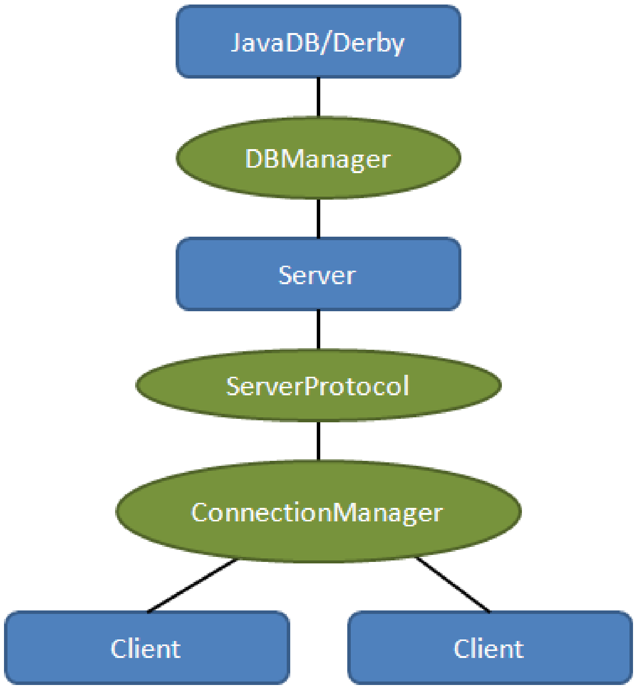
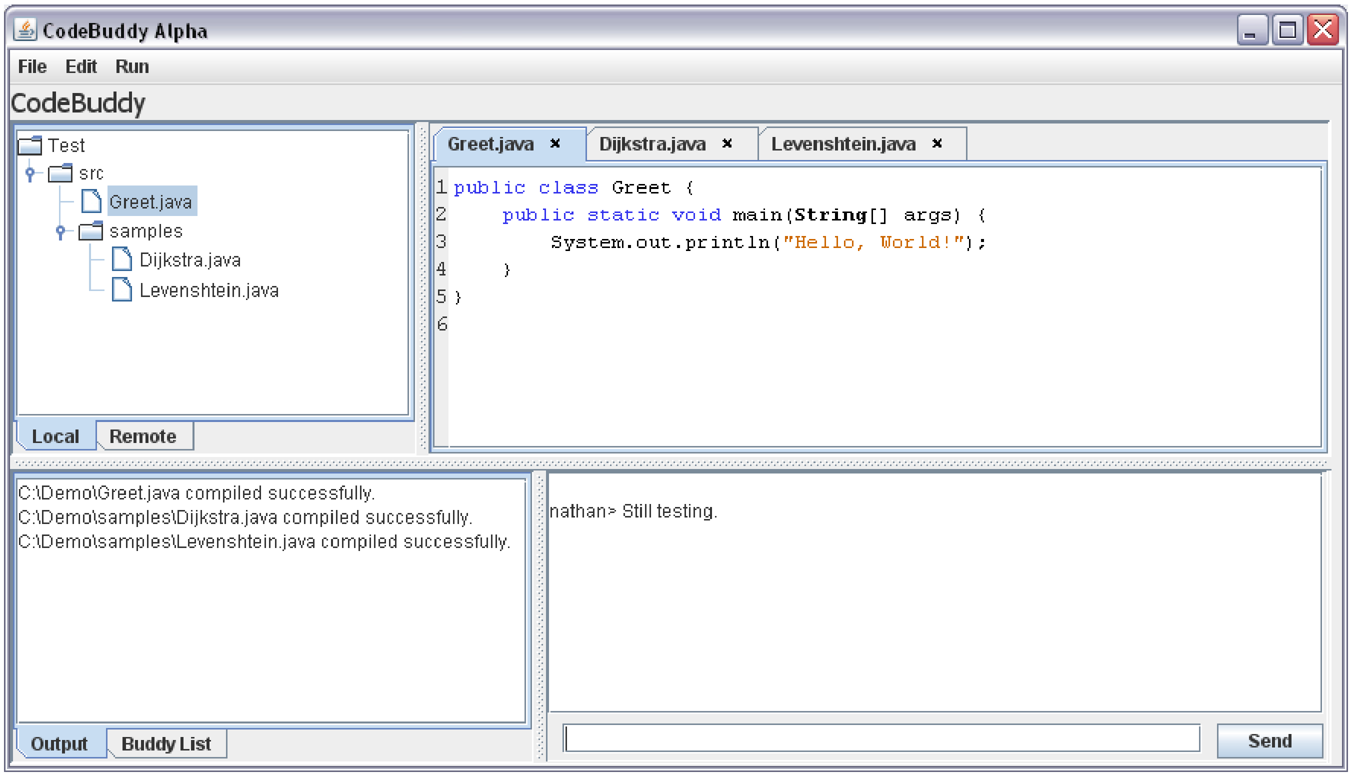


Figure 1. Overview of the high-level components used in the CodeBuddy system. JavaDB/Derby is an external package for maintaining the data on the backend; all other components are developed for the project.



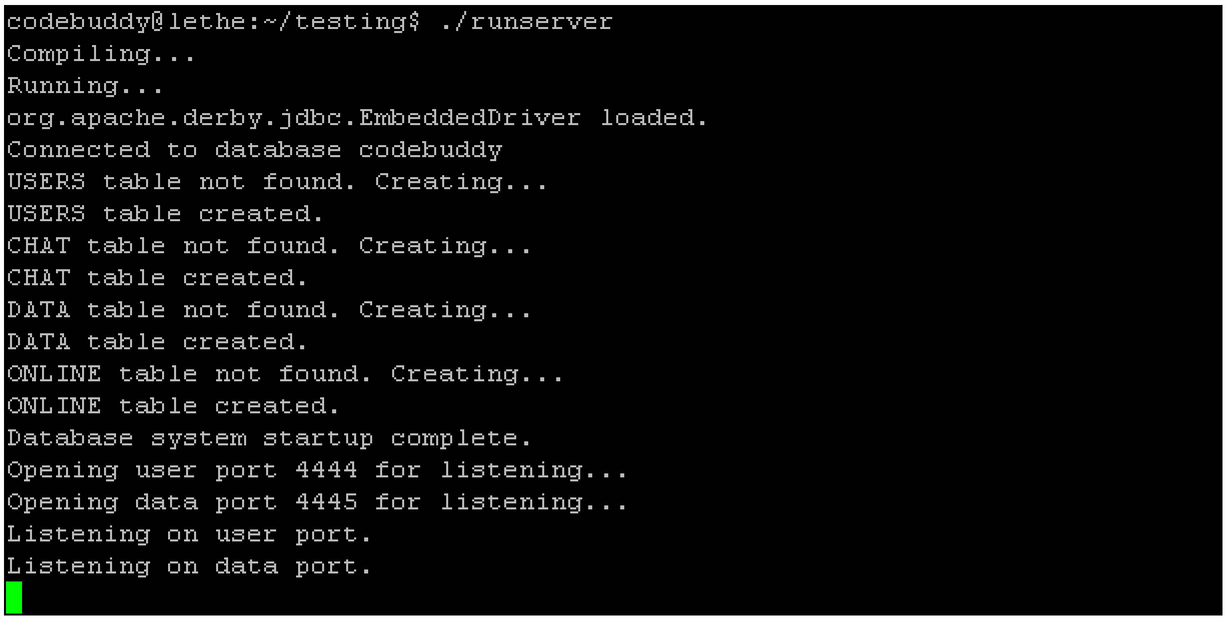
```java
public class Greet {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}

```

```
C:\Demo\Greet.java compiled successfully.
C:\Demo\samples\Dijkstra.java compiled successfully.
C:\Demo\samples\Levenshtein.java compiled successfully.
```

```
nathan> Still testing.
```

Figure 2. Primary client application view, showing a three-file active project in the top half, a successful in-program compilation attempt of the project in the bottom left view, and a test of the chat system in the bottom right view.



```
codebuddy@lethe:~/testing$ ./runserver
Compiling...
Running...
org.apache.derby.jdbc.EmbeddedDriver loaded.
Connected to database codebuddy
USERS table not found. Creating...
USERS table created.
CHAT table not found. Creating...
CHAT table created.
DATA table not found. Creating...
DATA table created.
ONLINE table not found. Creating...
ONLINE table created.
Database system startup complete.
Opening user port 4444 for listening...
Opening data port 4445 for listening...
Listening on user port.
Listening on data port.
```

Figure 3. Sample run of the server application, showing an annotation of the start-up process (server code compilation, Derby/JDBC start-up, and network listener start-up) via command line messages upon the first server start.

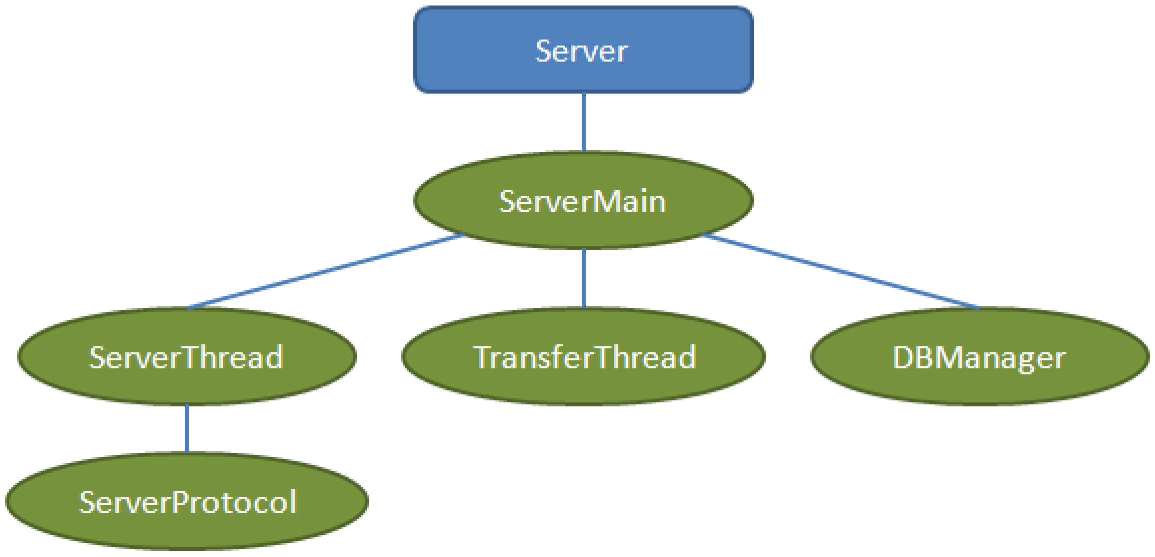


Figure 4. Schematic overview of the server code package. For each client connection, one ServerThread object is created, managing its connection state using the ServerProtocol class. The DBManager is a static class, shared among all threads, for interfacing with the JavaDB system for chats and authentication. TransferThreads are created for each client-to-server or server-to-client file transfer (downloading or uploading a code file).

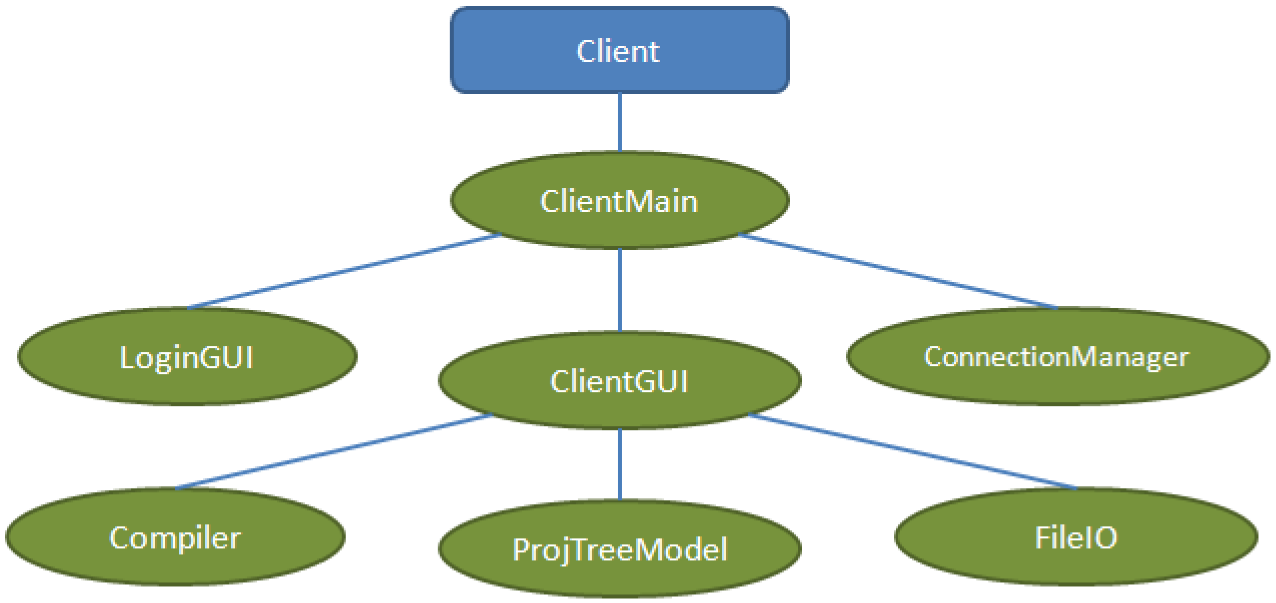


Figure 5. Schematic overview of the client code package. Each client starts with the LoginGUI, presenting the option to enter the program with his user account. Upon successful login, the ClientGUI is loaded, offering chat and editing functionality. The ProjTreeModel class displays project information; the FileIO class manages reading and writing code files; and the Compiler class handles in-editor code compilation and execution. All interactions with the server are handled exclusively by the ConnectionManager, which is configurable for different server setups. Minor classes for supporting project display, editor functionality, and handling user input are omitted for simplicity.