Online Development Platform
Molan IDE

A PROJECT REPORT PRESENTED
BY

PROGYAN BHATTACHARYA ROLL NO: 30000215015
DEEPAK THAKUR ROLL NO: 30000215006
PRAKASH KUMAR ROLL NO: 30000215014

TO
THE SOFTWARE ENGINEERING CURRICULUM

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
BACHELOR OF TECHNOLOGY
IN THE SUBJECT OF
INFORMATION TECHNOLOGY

MAULANA ABUL KALAM AZAD UNIVERSITY OF TECHNOLOGY
SALT LAKE, KOLKATA, WEST BENGAL
MAY 2018

Acknowledgments

This is an immense pleasure and gratification to produce our Software Engineering Project Report and acknowledging the indebtedness to all the persons and organizations helped for the same.
We are especially grateful to Dr. Suparna Biswas for giving us opportunity to work with such an interesting topic and providing us constant encouragement, support and valuable insides that helped us to carry out the idea successfully. We are also indebted to Mr. Ramesh Saha for his precious and utmost support and suggestions for the project development.

_____
Progyan Bhattacharya
(actg. Project Manager and Frontend Supervisor)

_____
Deepak Thakur
(actg. Project Coordinator and Designer)

_____
Prakash Kumar
(actg. Backend Supervisor)

Date: _____

Table of Contents

Software is a place where dreams are planted and nightmares harvested, an abstract, mystical swamp where terrible demons compete with magical panaceas, a world of werewolves and silver bullets.

Brad J.Cox

1 Overview

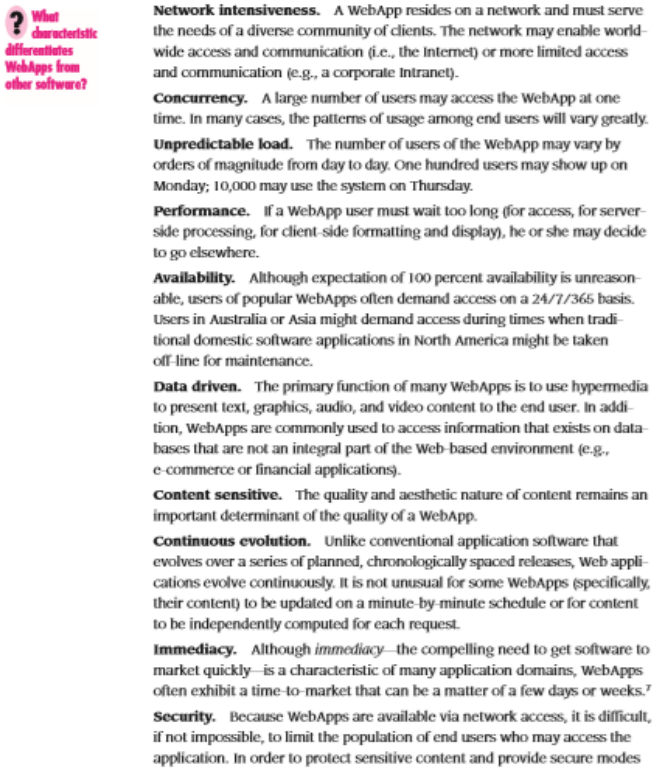
A good software is something that has been made with clear goal in mind, and by clear goal I mean a well specified *Software Requirement Specification*, well documented *Technology Stack* and a proper *Business Model* for marketing of the product.

Molan IDE is an online web based *Integrated Development Environment* with a flexible text editor and multiple language support and deployed on a Linux system. On layman's term, user writes the program in the space provided by the text editor after selecting the language, followed by the submission. And our product will build the program and return back the output, if success, error message otherwise.

This software is mainly targeted for Students (Grad or High-school), self-made

Programmers to make their learning process easy and guided. Developers can also use our product to write and test small modules before integrating to their large applications.

There are already a handful number of online IDE available in the market. Ideone *(by SPOJ)*, RunKit *(by Stack Exchange)*, jsFiddle just to name a few. But all of them have embedded a text editor on a static HTML causing the web application to be stagnant and less efficient with user interaction. Also their support for user helper features such as auto-completion, auto-bracing, build warning during development etc. is very much limited compared to desktop equivalent. What we target to build here is a scalable and efficient web based IDE that can give a proper competition to it's desktop equivalents. Also these features can be wrapped into responsive mobile (Android/iOS) application easily.



? What characteristic differentiates WebApps from other software?

Network intensiveness. A WebApp resides on a network and must serve the needs of a diverse community of clients. The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (e.g., a corporate Intranet).

Concurrency. A large number of users may access the WebApp at one time. In many cases, the patterns of usage among end users will vary greatly.

Unpredictable load. The number of users of the WebApp may vary by orders of magnitude from day to day. One hundred users may show up on Monday; 10,000 may use the system on Thursday.

Performance. If a WebApp user must wait too long (for access, for server-side processing, for client-side formatting and display), he or she may decide to go elsewhere.

Availability. Although expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a 24/7/365 basis. Users in Australia or Asia might demand access during times when traditional domestic software applications in North America might be taken off-line for maintenance.

Data driven. The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications).

Content sensitive. The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.

Continuous evolution. Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously. It is not unusual for some WebApps (specifically, their content) to be updated on a minute-by-minute schedule or for content to be independently computed for each request.

Immediacy. Although *immediacy*—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time-to-market that can be a matter of a few days or weeks.[7]

Security. Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to protect sensitive content and provide secure modes

Figure 1.0.1: Characteristics of Web Apps

If the process is right, the results will take care of themselves.

Takashi Osada

2

Software Model

In a fascinating book that provides an economist's view of software and software engineering, Howard Baetjer, Jr.[1], comments on the software process:

"Because software, like all capital, is embodied knowledge, and because that knowledge is initially dispersed, tacit, latent, and incomplete in large measure, software development is a social learning process. The process is a dialogue in which the knowledge that must become the software is brought together and embodied in the software. The process provides interaction between users and designers, between users and evolving tools, and between designers and evolving tools [technology]. It is an iterative process in which the evolving tool itself serves as the medium for communication, with each new round of the dialogue eliciting more useful knowledge from the people involved."

2.1 Process Model

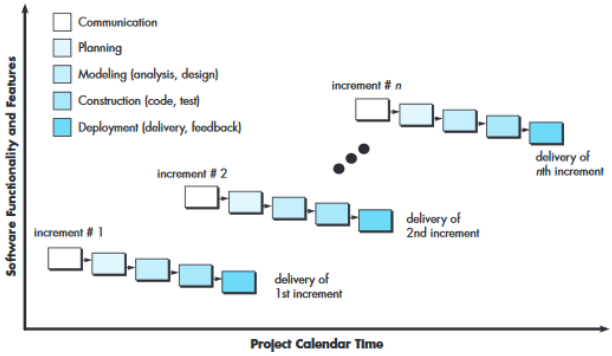


Figure 2.1.1: Incremental Model of Software Process

Every software production requires finite number of steps to be followed. In this project, we have followed *Incremental Prototyping Model*. Therefore, our development of product is based on independent features specification, instead of full software specification like *classical Waterfall Model*. In this chapter, we will discuss about how each increment is made plan for, designed, implemented and tested before moving to the next one.

2.2 Increments

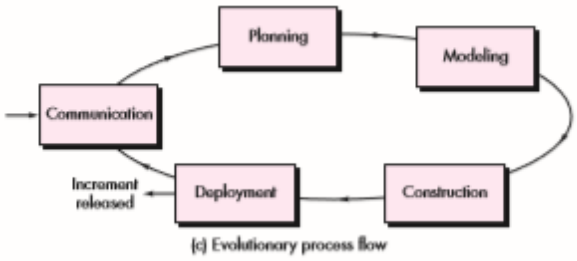


Figure 2.2.1: Evolutionary Model of Software Process

2.2.1 Environment Setup

The first phase of our product development is to choose a development platform, required tools and frameworks and make them ready for work.

- **Communication:** We discussed among each other and our project mentor about our idea of developing an web based *Integrated Development Environment*. We proposed to our mentor our project and got it accepted.
- **Planning:** We distributed the software broadly into two parts- a *Web Interface Application* and a *Representation State Application Program Interface*. We choose Reactive UI Component-driven technology to design our web interface, whereas we are using a micro HTTP service framework for API. For logging purpose, we decided using JSON file format.
- **Modeling and Construction:** We developed a minimal REST API service and Frontend interface to make sure our system works and by means of measuring their computational complexity and resource usage, we recognize *Hardware Requirements* for our product.
- **Deployment:** We deployed the application in *Red Hat Enterprise Linux* platform, installed required packages and tested for live status and performance.

2.2.2 Basic Layout

The very next stage is to design a basic layout for our application.

- **Communication and Planning:** We observed several existing product on same domain and revisited our requirements to come up with the vary basic layout that will be easy to use, easily scale and can hold all required features.
- **Modeling:** We model our application in a *Model-View-Controller* fashion. Contrary to the classical MVC architecture, our application has two

completely independent module communicating via HTTP. In our REST API service, the structure is made as *Model-Controller-API*, whereas the Frontend Web interface works in *Controller-View-Controller*.

- **Construction:** We developed a 3-tier application structure, i.e., one header, one footer and a main section. The header will have the product title and required navigation. The footer will display copyright notice and API status to make user stay updated. The main section will update it's content according to the user navigation.
- **Testing and Deployment:** We uploaded the program into server computer and tested for regression to make sure nothing broke by this patch.

2.2.3 Implement Editor

This one of the most vital phase of our application development.

- **Communication and Planning:** We specified the structure and it's interaction with user. Also, we defined the basic features for development environment, e.g., syntax highlighting, auto brace-enclosing, auto string-enclosing, auto indentation etc. to be present.
- **Modeling:** We defined an array of keywords and a click-event handler for Editor, that on user input will response accordingly.
- **Construction:** We create a dropdown select tool for language, selecting one will update the setup of the editor. The editor is rendered in the main section with dynamic size so that it can be viewed properly across all media size and devices.
- **Testing and Deployment:** We uploaded the program into server computer and tested for regression. Also we define a set of UI test against the changes made by this patch.

2.2.4 API Specification and Development

In this phase, we will structure our Database and API service to work along with web interface.

- **Communication and Planning:** We designed API specification, i.e, what are the data needed to perform an operation and what are the output produced. Both request and response are structure in *JavaScript Object Notation*.
- **Modeling:** The set of functionality are distributed among Models (that handles the data part) and Controllers (that handles the request part). The API server is structured to process incoming requests concurrently and asynchronously.
- **Construction:** For each incoming request, the API service will invoke a new *Shell* session and build the program passed as data. On successful completion, the output data will send back to the Controller for HTTP response; error message will be sent otherwise.
- **Testing and Deployment:** We uploaded the program into server computer and tested against sample programs and respective outputs as test cases.

2.2.5 Integrate UX with API

In this phase, we synchronize our frontend development with that of backend.

- **Communication and Planning:** We finalize API specification on previous step and decide to follow the same in Web Interface.
- **Modeling and Construction:** We create JSON object on Submit event (when user wants to submit the program to run on server) and Status event (user wants to check availability of server) according to the API specification and send them to respective *endpoints*.

- **Testing and Deployment:** We generate blackbox testing for UX functionality and utility functions. We deployed the work on server after being test successful.

2.2.6 User Registration and Log-in

In this phase, we design new entry to our database schema for user personal history.

- **Communication and Planning:** We discussed about user interaction with our application and decided to make a log entry for each user in our database..
- **Modeling:** Whenever user logs into the system, he/she can access all the details of his/her past interactions as well as last state of the text editor. This will enable user to continue development across multiple device.
- **Construction:** We create another entry in our database system holding tuples with *username, password (hash encrypted), log history and last program array*. We implemented and added to the existing API specification and bind it to UX.
- **Testing and Deployment:** We created a dummy user to check continuously if the log-in system is properly working and providing necessary security and privacy to it's user..

2.2.7 Local and Global Cache

This phase is continuation of previous development for user database.

- **Communication and Planning:** After we completed user registration and log-in system, we aimed to make text editor synchronize after a log-in or log-out event.

- **Modeling:** We defined an array of programs in both local storage and server database. When an user logs into the application, the global cache will be received as response.
- **Construction:** After receiving global cache as response, as per previous development, the application will compare it's local counterpart and select the one with latest timestamp and populate them on the text editor.
- **Testing and Deployment:** We uploaded the program into server computer and did integration test. Also make sure it does not fail any previous test suite.

The hardest single part of building a software system is deciding what to build.No part of the work so cripples the resulting system if done wrong.No other part is more difficult to rectify later.

Fred Brooks

3

Requirement Specification

Understanding the requirements of a problem is among the most difficult tasks that face a software engineer. Although it may seem that developing a clear understanding of requirements is not hard, as if the customer does not know what is required, end users does not have a good understanding of the features and functions that will provide benefit. Surprisingly, in many instances these situations are not only possible but *frequent*. And even if customers and end-users are explicit in their needs, those needs will change throughout the project.

In the forward to a book by Ralph Young[2] on effective requirements practices, Roger E. Pressman wrote:

It's your worst nightmare. A customer walks into your office, sits down, looks you straight in the eye, and says, "I know you think you understand what I said, but what you don't understand is what I said is not what I meant." Invariably, this happens late

in the project, after deadline commitments have been made, reputations are on the line, and serious money is at stake.

3.1 Hardware Requirements

3.1.1 Server

- **Processor:** Quad-core i3 or equivalent processor @2.00 GHz
- **Physical Memory:** Minimum 4 Gigabyte
- **Network:** 10-100 GBps Bandwidth

3.1.2 Client

- **Processor:** Pentium or above @1.60 GHz
- **Physical Memory:** Minimum 1 Gigabyte
- **Network:** 1-100 MBps Bandwidth

3.2 Software Requirements

3.2.1 Server

- **Operating System:** Red Hat Enterprise Linux or any other variant
- **Build Utility:** GNU C/C++ Build-util, Python 3.x IDLE, Python Virtual Environment, Node.JS 8.x LTE
- **Package Manager:** Pip (Python 3), Npm
- **Package Dependency:** Requirements.txt, Package.json

3.2.2 Client

- **Operating System:** Any operating system with networking capability
- **Utility:** Web Browser (script-enabled)

3.3 Design and Implementation Constraint

Key constraint of this product is the usage of local memory and the network bandwidth. The bundled software needs to be as small/minified as possible so that it can be easily transferred via network protocols. Also the way of development is made asynchronous in nature so that, when an action is performed the user interface will stay responsive at any case. Since the service is broken into two pieces, i.e., a REST API service and a full fledged Web Application communicating via HTTP requests, the network bandwidth creates the bottleneck in the performance.

3.4 Assumptions and Dependencies

- The browser in which the application runs must have proper permission to use internet.
- The browser supports scripting functionality, i.e., JavaScript methods and prototypes.
- The browser have necessary features and permission to create asynchronous XHR calls to *Hypertext Transfer Protocol*.
- The browser can send HTTP requests to different origin, i.e., *CORS (Cross-Origin Resource Sharing)* enabled. The browser have proper support for session, cookies and local storage required by the application. The development utilities that have been used as framework of the application will update itself maintaining backward compatibility.External Interface Requirements

3.5 Terms of Use

GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The most common miracle of software engineering is the transition from analysis to design and design to code.

Richard Due'

4

Software Design

Designing a software, is the most crucial job of a software engineer. A well designed software not only find it's use and demand on user base but provides life long opportunity to make it better manageable and serviceable.

4.1 Features

After all requirement gathering and analysis, the software is finalized to offer the following features to end users:

- **Reactive Text Editor:** A user command sensitive text editor for writing the program. The editor will also provide basic assistance to the users by *syntax highlighting*, *auto-brace closing*, *auto-indentation* etc.

- **Language Option:** A pool of programming language will be offered to the user so that user have the freedom to choose which language to user for their program.
- **File Handling:** The application will let the user upload the program to the editor from a file residing in local storage and vice-versa.
- **Boilerplate Template Code:** The application will offer users a minimal boilerplate code for each language to increase their development pace.
- **Reset Formatting:** A quick button to reset all the codes written into the text editor and reset back to basic templates.
- **Build Procedure:** The application will include a set of build procedure and output generation technique for each programming language and will invoke the same when a program is submitted.
- **User Authentication**: A prime user will register and log-in to the system to get available features and share the work across multiple devices.
- **Cache Support:** The application will offer both local *(user machine)* and global *(server database)* cache in order to make developers able to continue from where they left. Each time an user logs into the system, local cache gets updated by global cache, if it is more recent.

4.2 Class Diagram

Class Diagram helps us to understand the abstract skeleton of the data objects used by the application. In our application, we mainly deal with *Programs* and *Users*.

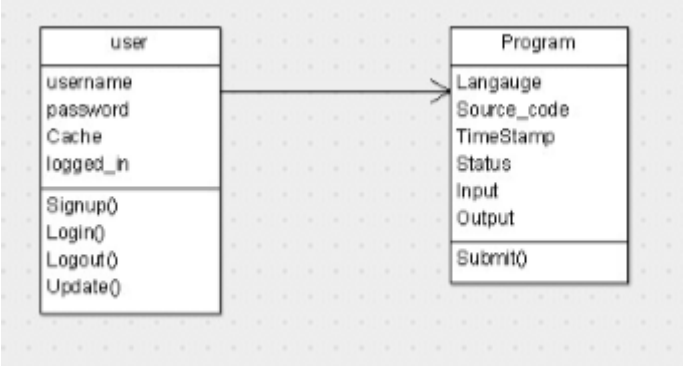


Figure 4.2.1: Class Diagram

4.3 Use-case Diagram

Use-case diagram is a short-hand for listing out the feature the application offers to the end users and how their actions are transferred to the server or any 3rd party involved.

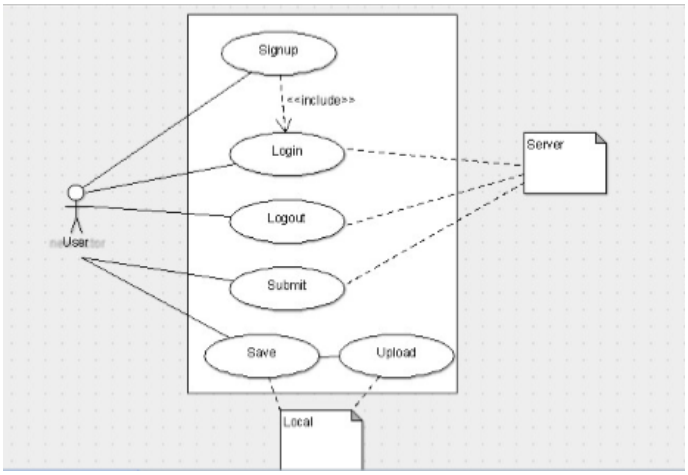


Figure 4.3.1: Use-case Diagram

4.4 Sequence Diagram

Sequence Diagram defines the flow of control through multiple component or module of the application. The sequence of actions for this particular application can be broadly specified into two categories: *(a) Regular or non-Registered Users* and *(b) Premium or Registered Users*.

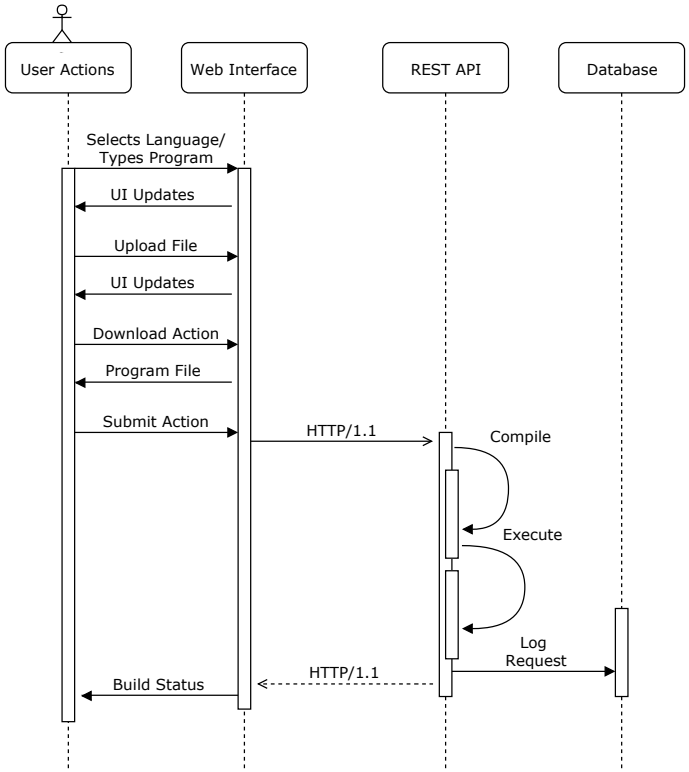


Figure 4.4.1: Sequence Diagram - Regular User

On top of every features offered to regular users, premium users are entertained with the concept of *global cache*, i.e., storing their development into database so that they can access them across all devices.

4.5 Data-flow Diagram

Data-flow Diagram denotes the flow or the architecture of data manipulation by the software. Here, we've shown upto *level 3* of data-flow in our application.

4.5.1 Level 0 Data-flow Diagram

Here *input* can be defined as Language-Program pair and *output* can be considered as Status-IO pair.

Figure 4.5.1: Data Flow Diagram - Level 0

4.5.2 Level 1 Data-flow Diagram

The application is based on *two-tier architecture* using one web interface and a REST API service.

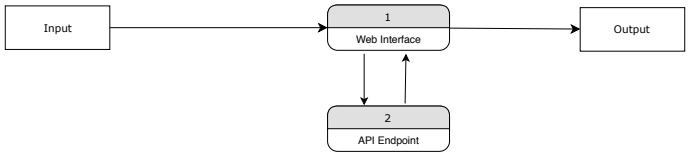


Figure 4.5.2: Data Flow Diagram - Level 1

4.5.3 Level 2 Data-flow Diagram

This diagram gives an insight on how the API manages the data from incoming HTTP requests and generate appropriate respose to them.

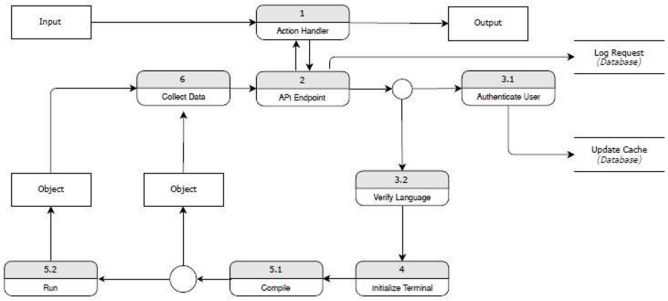


Figure 4.5.3: Data Flow Diagram - Level 2

4.6 Component Diagram

Component-level Design gives an essence of how modules are designed, implemented and connected to each other to form the application. The component diagram shown below briefly describes the components used in web interface of our applications.

4.6.1 Cohesion between Components

In computer programming, cohesion refers to the degree to which the elements inside a module belong together. In one sense, it is a measure of the strength of relationship between the methods and data of a class and some unifying purpose or concept served by that class. In another sense, it is a measure of the strength of relationship between the class's methods and data themselves.

4.6.2 Coupling between Components

In software engineering, coupling is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules.

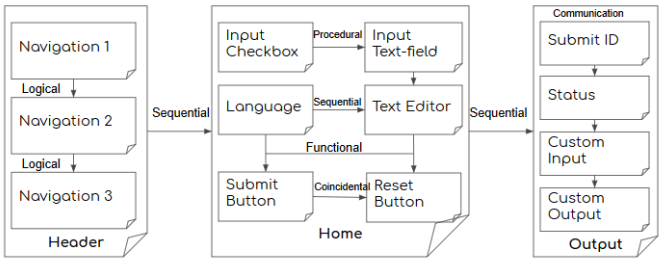


Figure 4.6.1: Component Level Design - Cohesion

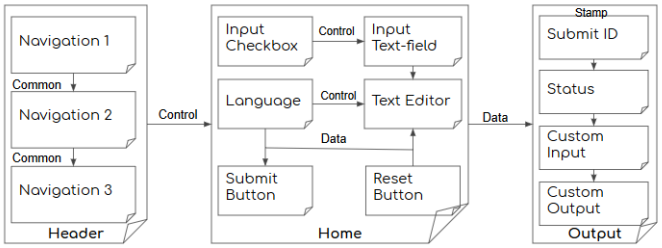


Figure 4.6.2: Component Level Design - Coupling

4.7 Test Suites

4.7.1 Black-box Testing

All the UX functionality along with API specification are being properly tested before deployment. This test includes but not limits to collection of data objects, forming HTTP requests, sending asynchronous request to API and receiving the responses, mutating state of application accordingly.

4.7.2 White-box Testing

All UI components are being tested as well. During this test, the machine looks for all the components to be rendered at particular position, handling user inputs and calling event handler.

In an age of outsourcing and increased competition, the ability to estimate more accurately... has emerged as a critical success factor for many IT groups.

Rob Thomsett

5 Business Model

A SOFTWARE IS ESSENTIALLY A PRODUCT; and like any other product, a successful software requires a strong business model. In this last chapter of the project report, we will discuss about the user base of the application, revenue program and scope of improvements.

5.1 Intended Audience

Our main user base contains Students or self-taught Programmers, developers and Academic Institutions.

- For Students they can practice their coding skills. And with help of this product they can understand their errors more accurately.

- For Developer (Professional) can verify their design goals, build small functionality and associated tests which will help them to build applications in faster pace.
- For Institution they can take their examination or any programming challenges using prime features.

5.2 Revenue Model

The software can be used as a medium for learning programs and our revenue models is based upon the assumption that if an enterprise, organization or institution buys our product and service, this will help their clients or students respectively to be more productive in nature.

For example, an university can buy the product to make their technical evaluations more fair, error-free and fluid, causing ease of reporting. An enterprise, on the other hand, will be able provide a platform to it's developers to test their designs before starting any project.

5.3 Room for Improvement

Due to short-time development and relatively less experience, the project is still in it's beginner phase and demands to be prosper. Following is some few options that can improve the attraction and quality of service offered by the application:

- **Plagiarism Checker:** The application when running in examination/competition mode, will check for copied programs with high precision.
- **Scoring Technique:** The application should score the participants according to the accuracy, memory usage, execution time and time of submission. There should be an *Admin Panel* that will allow user to generate and upload mark sheet.

References

[1] Howard Jr. Baetjer. Software as capital. *IEEE Computer Society Press*, page 85, 1998.

[2] R. Young. Effective requirements practices. *Addison-Wesley*, 2001.