BPjs Online IDE
Amir Dimri | Guy Shuster | Ofek Ashkenazi

Application Design Document

BPjs Online IDE

Chapter 1 - Use Cases

1.1

Use Case Name: Editing code

Actors:

- System user

Trigger:

- The user puts the cursor on to the main code window or to the external event console

Preconditions:

- There is a network connection and the page was loaded properly

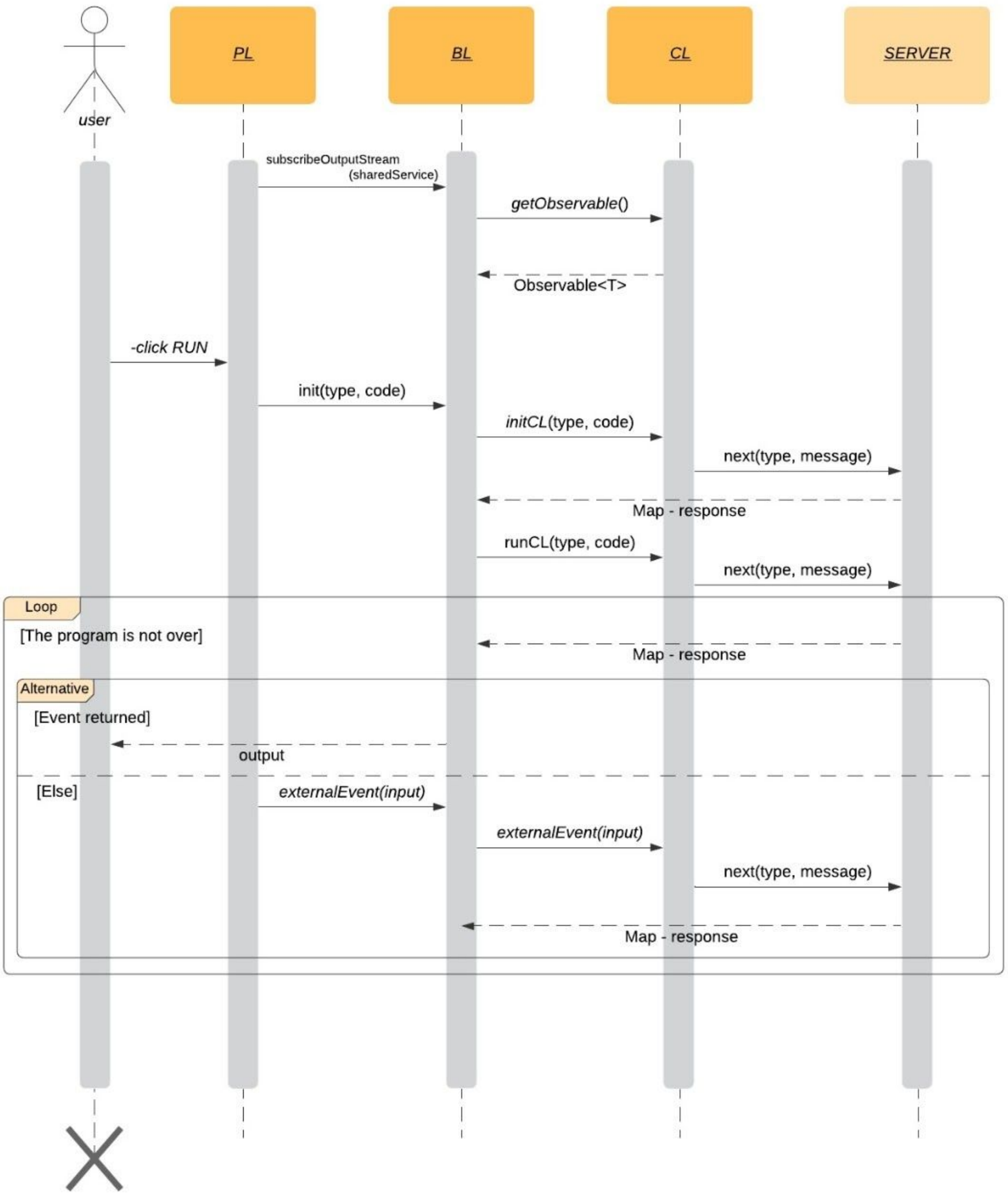
Postconditions:

- None

Normal Flow:

1. The user opens the browser and enters the BPjsOnlineIDE website.
2. The user puts the cursor on to the main code window or to the external event console
3. The user can now write and edit the code as he wishes

1.2 Executing code:



__Use Case Name:__ **Executing Code**

__Actors:__

- **System user**

__Trigger:__ **The user presses the “run code” button.**

__Preconditions:__

- **The user wrote some code.**

__Postconditions:__

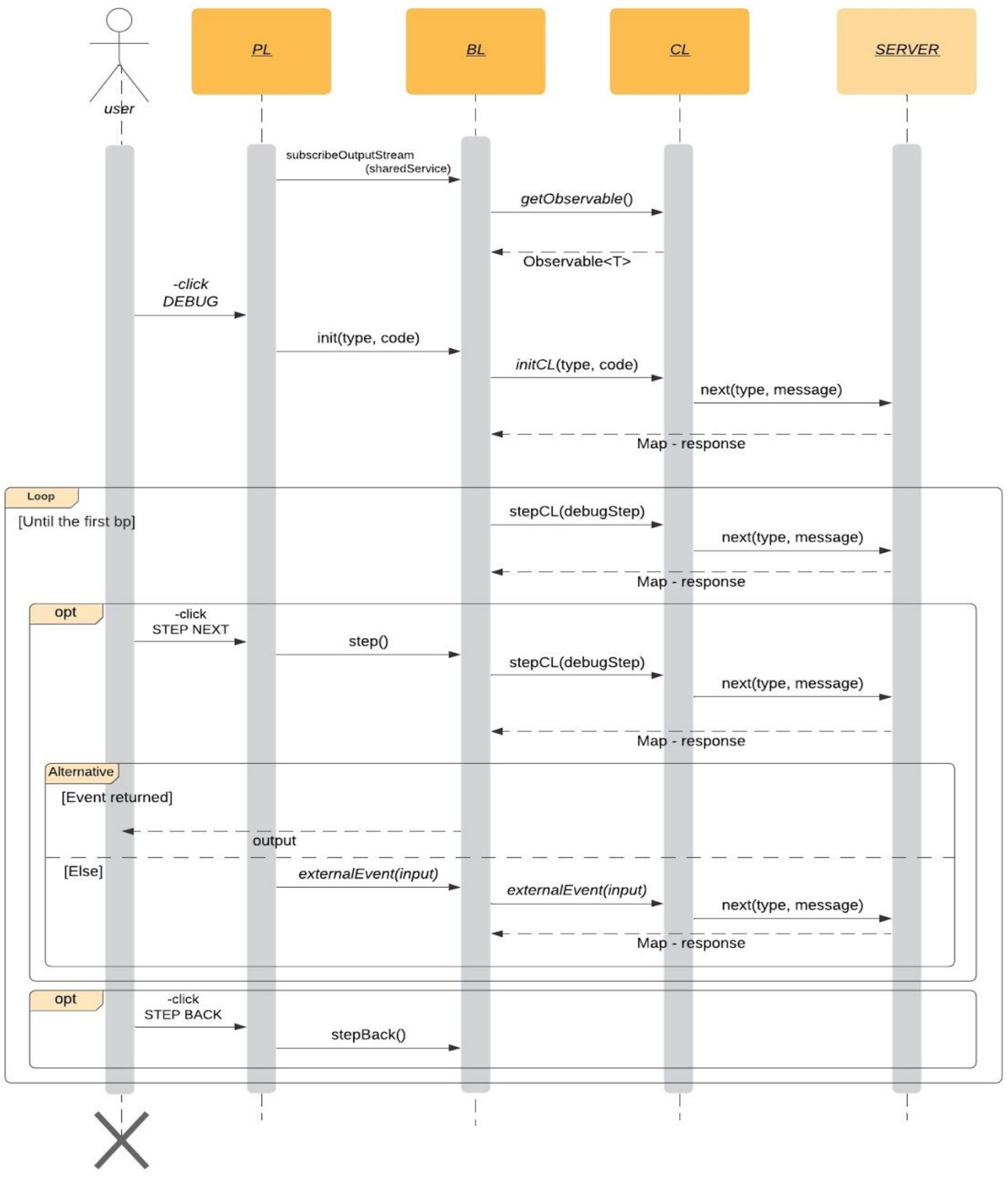
- **The code has been compiled.**

__Normal Flow:__

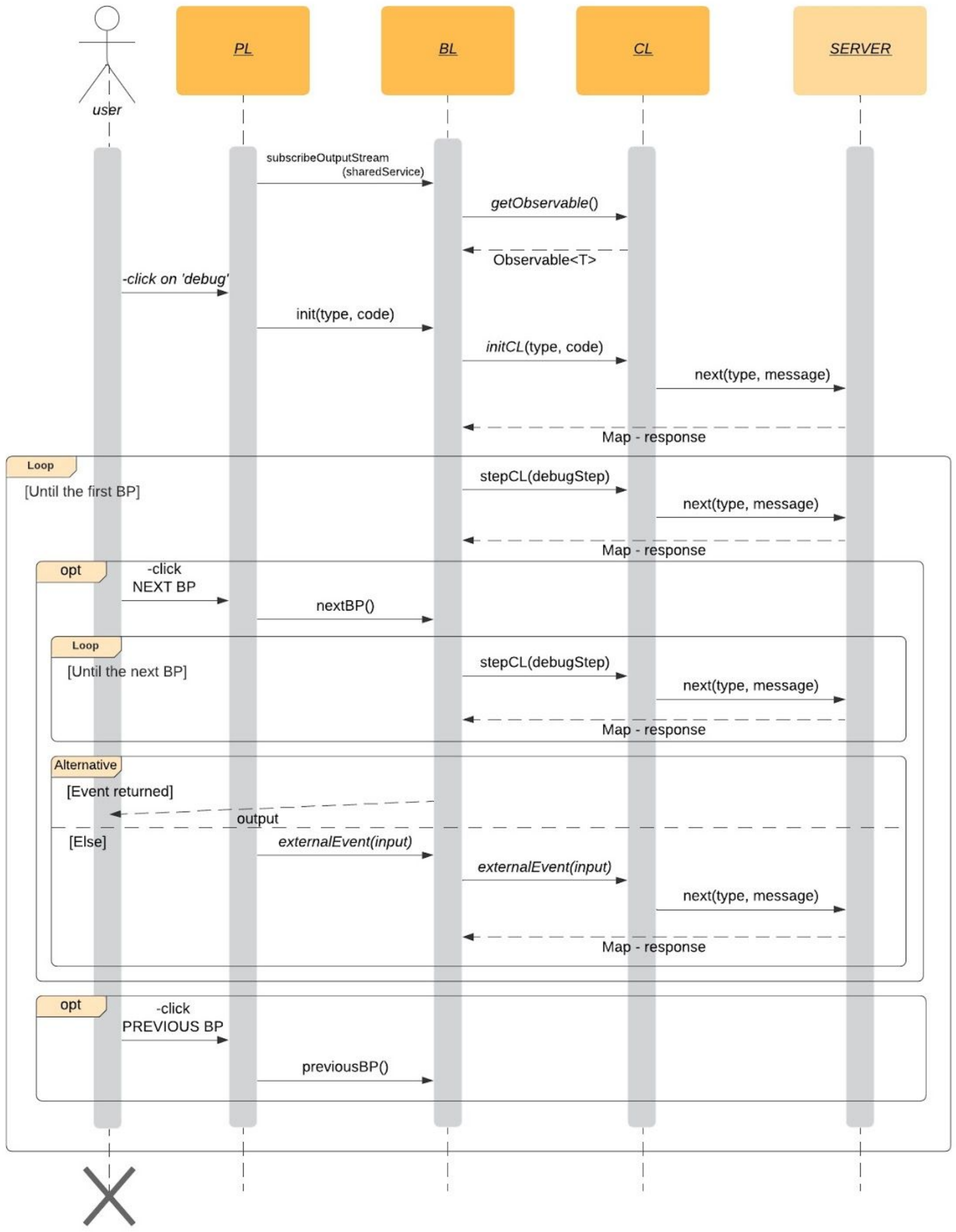
1. **The system user has written code that he wants to run, and maybe even added external events in the event console.**
2. **The system user runs the code he wrote.**
3. **The output of the program is shown in the output console.**

1.3 Debugging code:

Next Step \ Back Step:



Next BreakPoint \ Previous BreakPoint:



__Use Case Name:__ Debugging code

__Actors:__

- System user

__Trigger:__

- The user presses the “debug code” button

__Preconditions:__

- The user wrote some code and set a breakpoint

__Postconditions:__

- The user is able to see the values of variables of different b-threads, global variables, states of b-threads, a trace of the program thus far and all current events divided into groups (blocked \ not blocked).

__Normal Flow:__

1. The system user types code to be executed in the appropriate place.
2. The system user sets one or more breakpoints, and presses the Debug button
3. The window switches to debug display (more on this in chapter 5)
4. The program stops at the first breakpoint and the user is able to see the data described above, and also debug his code with the following functions:

   __Step:__
   Each press on the step button will make the code jump from the current bp.sync to the next one, without considering extra breakpoints.

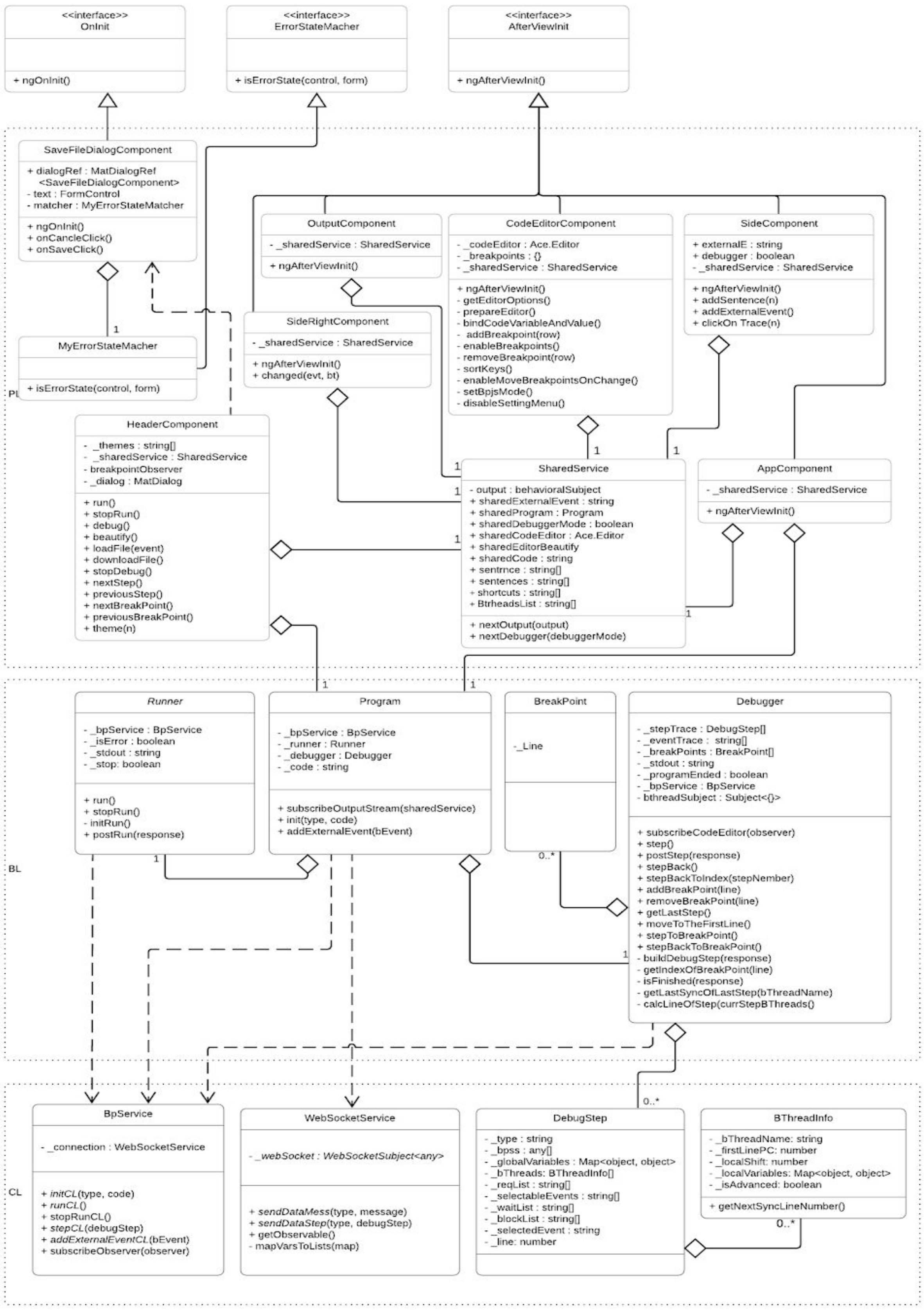
Continue:
Each press on the continue button will make the code jump from the current breakpoint to the next one (if present). If there is no breakpoint ahead, the current run will terminate and it’s result will be displayed in the output window.

Terminate:
Finishes the current run of the program, with the output displayed in the output window.

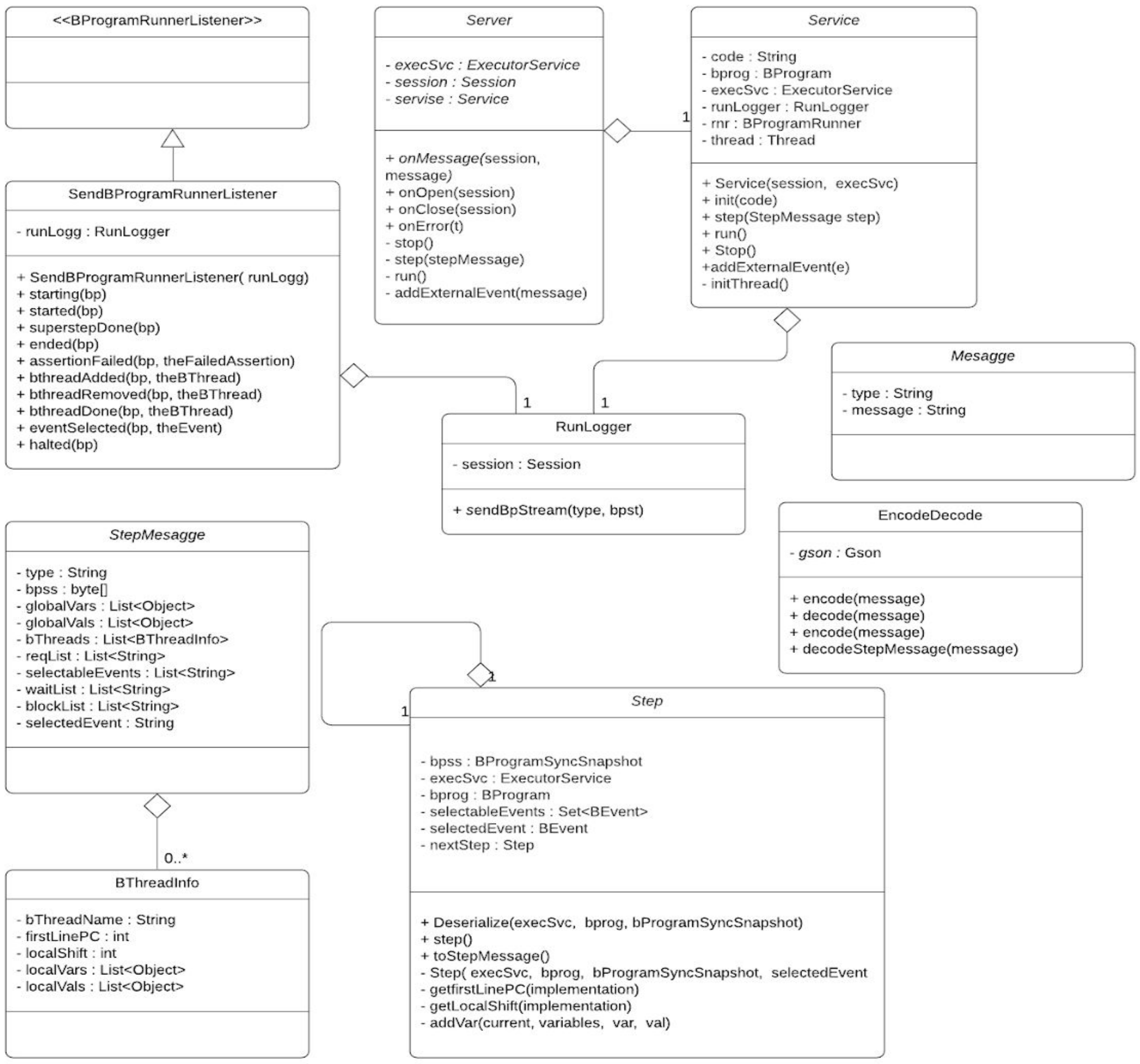
Alternate Flow:

If the user doesn’t set a breakpoint, the program will run until its end.

Class Diagram - Client:

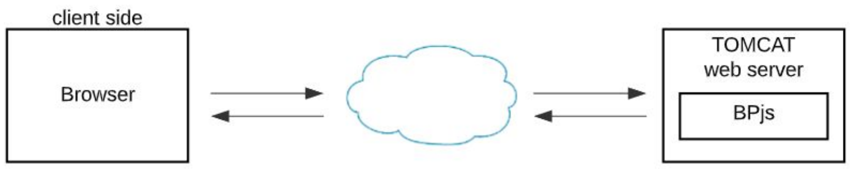


Class Diagram - Server:



<<BProgramRunnerListener>>
SendBProgramRunnerListener
- runLogg : RunLogger
+ SendBProgramRunnerListener(runLogg)
+ starting(bp)
+ started(bp)
+ superstepDone(bp)
+ ended(bp)
+ assertionFailed(bp, theFailedAssertion)
+ bthreadAdded(bp, theBThread)
+ bthreadRemoved(bp, theBThread)
+ bthreadDone(bp, theBThread)
+ eventSelected(bp, theEvent)
+ halted(bp)
Server
- execSvc : ExecutorService
- session : Session
- servise : Service
+ onMessage(session, message)
+ onOpen(session)
+ onClose(session)
+ onError(t)
- stop()
- step(stepMessage)
- run()
- addExternalEvent(message)
1
Service
- code : String
- bprog : BProgram
- execSvc : ExecutorService
- runLogger : RunLogger
- rnr : BProgramRunner
- thread : Thread
+ Service(session, execSvc)
+ init(code)
+ step(StepMessage step)
+ run()
+ Stop()
+addExternalEvent(e)
- initThread()
1
1
RunLogger
- session : Session
+ sendBpStream(type, bpst)
Mesagge
- type : String
- message : String
EncodeDecode
- gson : Gson
+ encode(message)
+ decode(message)
+ encode(message)
+ decodeStepMessage(message)
StepMesagge
- type : String
- bpss : byte[]
- globalVars : List<Object>
- globalVals : List<Object>
- bThreads : List<BThreadInfo>
- reqList : List<String>
- selectableEvents : List<String>
- waitList : List<String>
- blockList : List<String>
- selectedEvent : String
0..*
BThreadInfo
- bThreadName : String
- firstLinePC : int
- localShift : int
- localVars : List<Object>
- localVals : List<Object>
1
1
Step
- bpss : BProgramSyncSnapshot
- execSvc : ExecutorService
- bprog : BProgram
- selectableEvents : Set<BEvent>
- selectedEvent : BEvent
- nextStep : Step
+ Deserialize(execSvc, bprog, bProgramSyncSnapshot)
+ step()
+ toStepMessage()
- Step(execSvc, bprog, bProgramSyncSnapshot, selectedEvent
- getfirstLinePC(implementation)
- getLocalShift(implementation)
- addVar(current, variables, var, val)

Chapter 2 - System Architecture



There are a few major components to our system, which will be implemented as a Client - Server architecture. The client will be the web based IDE itself, with all included features, and the server will be the BPjs compiler and debugger. In this architecture, when the user will compile the code, the code will be sent to the server and compile and run there, and the output will be sent back to the client side. However, the other features, like syntax coloring or beautify or replace all etc., will be performed in the client side, without an intervention of the server side.

The system consists of 3 layers:
1. Presentation Layer - responsible for giving the user access to the system. It includes UI which includes several parts:

- **The central part - the text box where the developer will write the code to execute.**
- **Bottom of the screen - after executing a code, the output console in the bottom of the screen will show the output of the program. In debug view, the bottom of the screen will change and display also relevant information for the program run.**
- **On the left side of the screen there will be a Tool-Box that will include basic sentences in BPjs. A click on one of them will enter the specific sentence to the Text-Box and allow to the developer to create a code structure in BPjs. In debug view, the left side of the screen will swap and display relevant information for the program run.**
- **On the right side of the screen there will be a BThreads table - display only in debug mode.**
- **At the top of the screen there will be buttons that will enable different actions such as: arranging indentations, running the code, stop the run, uploading a file to the system, etc.**

2. Business Layer - This layer will store some data structures that will store information about the events that occur in the BPJS code (blocked, waiting, requested).

3. Communication Layer - A layer that links the system to the server that includes the BPjs compiler.

Chapter 4

Object-Oriented Analysis

4.2 Class Description

Client - Presentation Layer:

The main class in the presentation layer is the AppComponent class.
This class provides the functionality for the main component that includes our entire system.
It has methods like “runCode()”, “beautifyCode()” “theme()” etc., witch are connected directly to the HTML of the main component.

Client - Business Layer:

The main classes in the business layer are Debugger, Runner, Program and BreakPoint.
Debugger holds stepTrace and eventTrace which are parameters necessary for debugging the program, and has a step() method which jumps between bsync’s of the bThreads.
Program has a method init(type, code) which is called with the current user code to initiate a session with the server.
BreakPoint is a class which surprisingly enough defines a break point and all it’s relevant data in the user’s code.

4.4 Unit Testing

We created a GoogleShits document that centers all the unit tests.

For each unit test, the above data is listed:

Req id	Req desc	func	Test id	Test desc	Is done?

Link To The Full Test Tables

Chapter 5 - User Interface Draft

Our system has 2 main views:

1. **Working display:**

   In this mode the programmer can write code, run code, see the output of the program and perform various actions like: beautify, change the theme of the text box, etc.

   Working display consists of several parts:

   At the top of the screen there will be buttons that will enable different actions such as: arranging indentations, running the code, beautify the code, uploading a file to the system, etc.

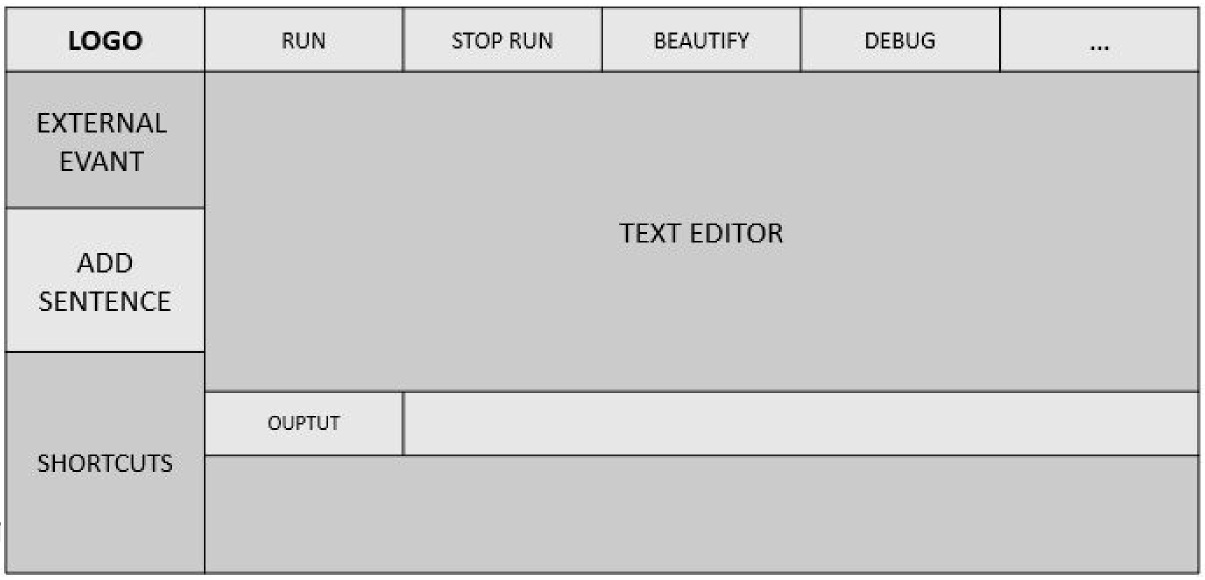
   Middle of the screen - the text box where the developer will write the code to execute.

   Bottom of the screen - after executing a code, the bottom of the program will show the output of the program.

   On the left side of the screen will be a Tool-Box that will include base sentences in BPjs that click on each of them will enter the specific sentence to Text-Box and allow to the developer to create a code structure in BPjs.

   In addition, on the left side of the screen will be a text-box - to be used for input for external events (a BPjs feature).

   Furthermore, a list of possible keyboard shortcuts for use in a code editor will appear on bottom of the left side.



2. <u>**Debugger display:**</u>

In this mode, the view will focus on the code execution step by step in favor of a more focused view of the code and its variables and values in each row in the program run.

At each stage the user can see what BTs are in wait \ block \ request and see in the code who BTs are currently running.

Debugger display consists of several parts:

At the top of the screen there will be buttons that will enable various actions such as: stop debugging, step into, etc.

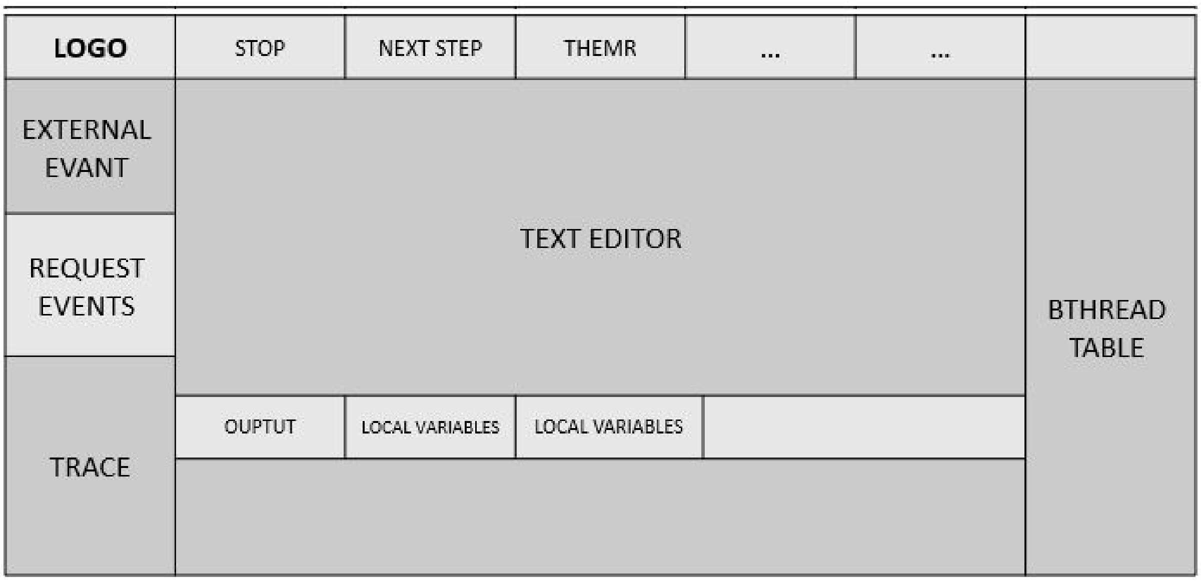
Middle of the screen - the text box which contains the BPjs code, which was run by the user.

Bottom of the screen - output of the program and 2 tables for local variables and global variables.

On the left side of the screen will be a three parts:

A. **text box - to be used for input for external events (a BPjs feature).**
B. **List of requested BTs, and each one's state (block \ not block).**
C. **The trace of the program. (BT list is in order of their running).**

On the right side of the screen will be a BThread table.



Chapter 6 - Testing Non Functional Requirements

Implementation Constraints:

The only thing left to test as far as implementation goes, is that the system is deployable to a cloud service/remote server as listed in the ARD.
Since we designed the frontend with Angular, and the backend runs on a tomcat server, the app can be deployed and placed in the "webapps" folder on the JAX-RS Java project, in a few simple steps. The technologies we chose to design our system are the test itself in this case.

UI Constraints:

We are constantly trying to fulfill UI constraints, because we are working with the Clarity Design System and Angular Material.
These are both open source design system that bring together UX guidelines, an HTML/CSS framework, and Angular components, and provide a set of well-designed and implemented data-bound components built on top of Angular - exactly what we need.
Their design is reactive and modern, and lives up to the requirements we defined in the ARD.