DELVE

User Guide

Contents

1. Introduction

Delve is a software tool designed for programming course instructors. It can analyze programs written in Java, aggregate the results and present an overview about the class performance in any given assignment. Although Delve is intended for CS1 and CS2 courses it can be used in any programming-oriented course.

Although Delve can analyze just the final submission, its strength lies on analyzing the program as it progressed over time, from start to final submission. Therefore, it is recommended that you collect every program variation over time (referred from now on as 'code snapshots' or simply 'snapshots') for every student and put them in a folder. To avoid conflicts in the file names, each snapshot uses the following naming convention:

<student ID>_<program name>_<snapshot number; ascending order>.java

For example, if student A produced 40 snapshots for a program Foo.java, the snapshots should be named:

a_Foo_01.java (first snapshot)
a_Foo_02.java
...
a_Foo_40.java (last snapshot/submitted program)

All snapshots can be dumped under the same directory. Delve will automatically sort them to subfolders (one subfolder per student who submitted a program).

Delve can perform a time analysis and determine how long it took to complete the assignment. To do so, each snapshot should be accompanied with a timestamp. The snapshot timestamps appear in file **timestamps.txt**. Each line in this file has the following format:

<snapshot name>#<timestamp in sec>

The timestamp must be expressed in seconds. Usually the timestamp is the difference, measured in seconds, between the snapshot creation time and midnight, January 1, 1970 UTC. However, it can be anything as long as it uses the first snapshot as reference.

For example. both:

```
foo_Bar_01.java#1609470315
foo_Bar_02.java#1609470326
...
foo_Bar_20.java#1609471626
```

and

```
foo_Bar_01.java#0
foo_Bar_02.java#11
...
foo_Bar_20.java#1311
```

are the same.

Similar to the snapshots which can be dumped in the same folder, there can b a single **`timestamps.txt`** with all snapshot timestamps, one per line, and in any order.

Delve will find automatically the timestamps that correspond to the snapshots of a given student and create a file **`timestamps.txt`** in the student's subfolder.

The following sections will guide you step-by-step how to use Delve.

2. Loading the snapshots to analyze

When Delve first launches the following window shows up:

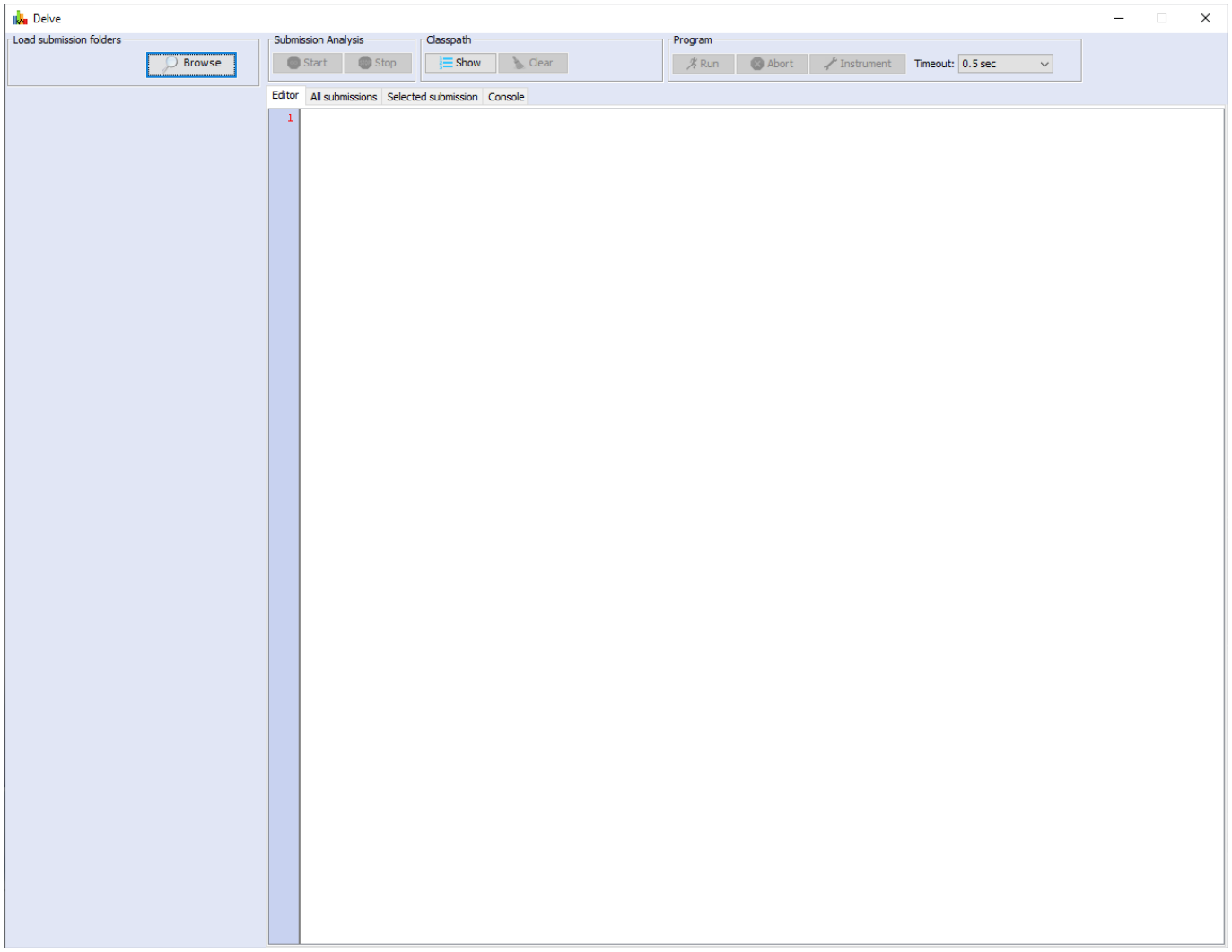

Figure 1: Delve upon launch.

Now, the user must select the root directory with the student snapshots. The root directory does not necessarily have to be organized (i.e., individual subfolders with the code snapshots and timestamps for every student). Delve will do that.

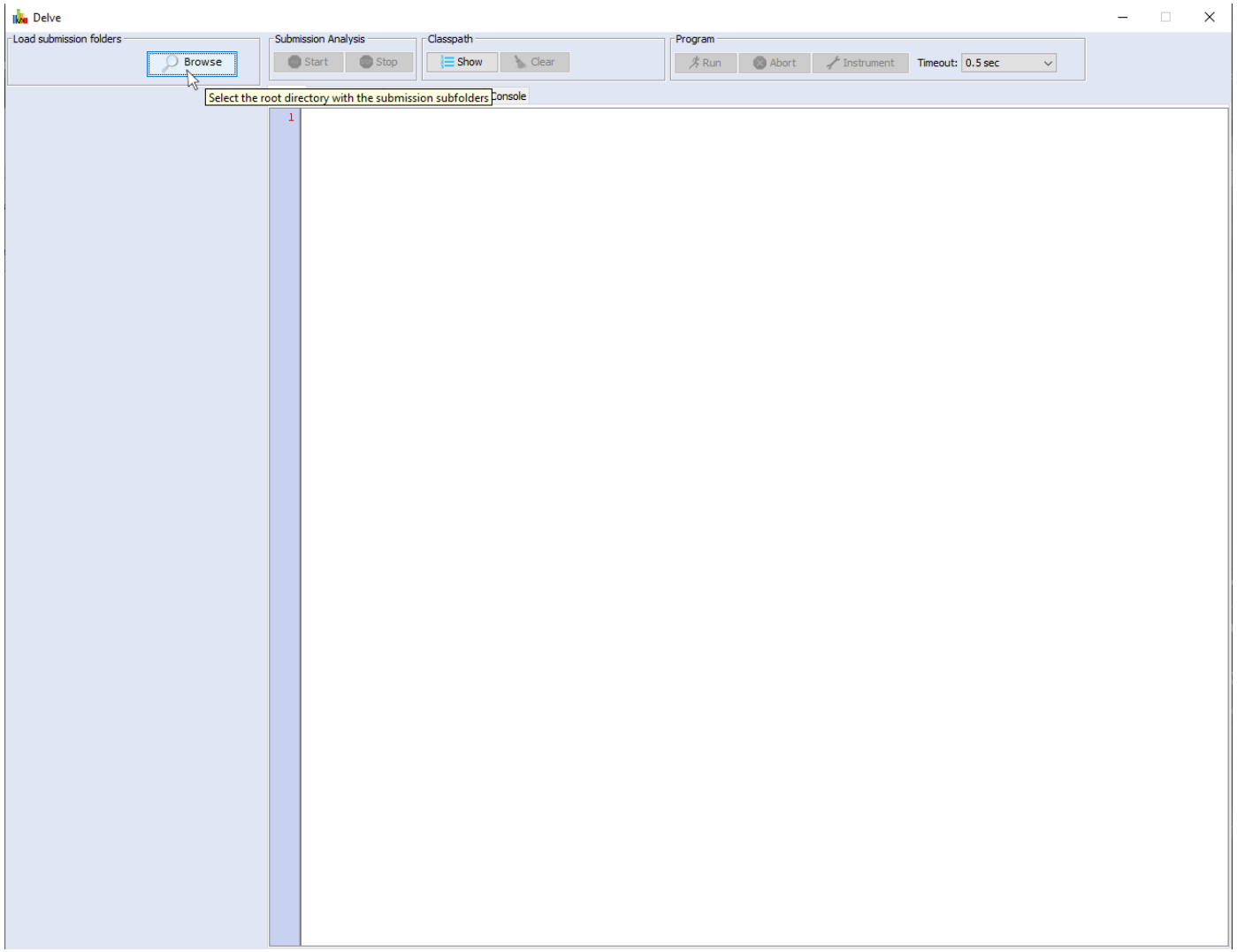

Figure 2: The user needs to click first on the *Browse* button to locate the root directory with the student code snapshots.

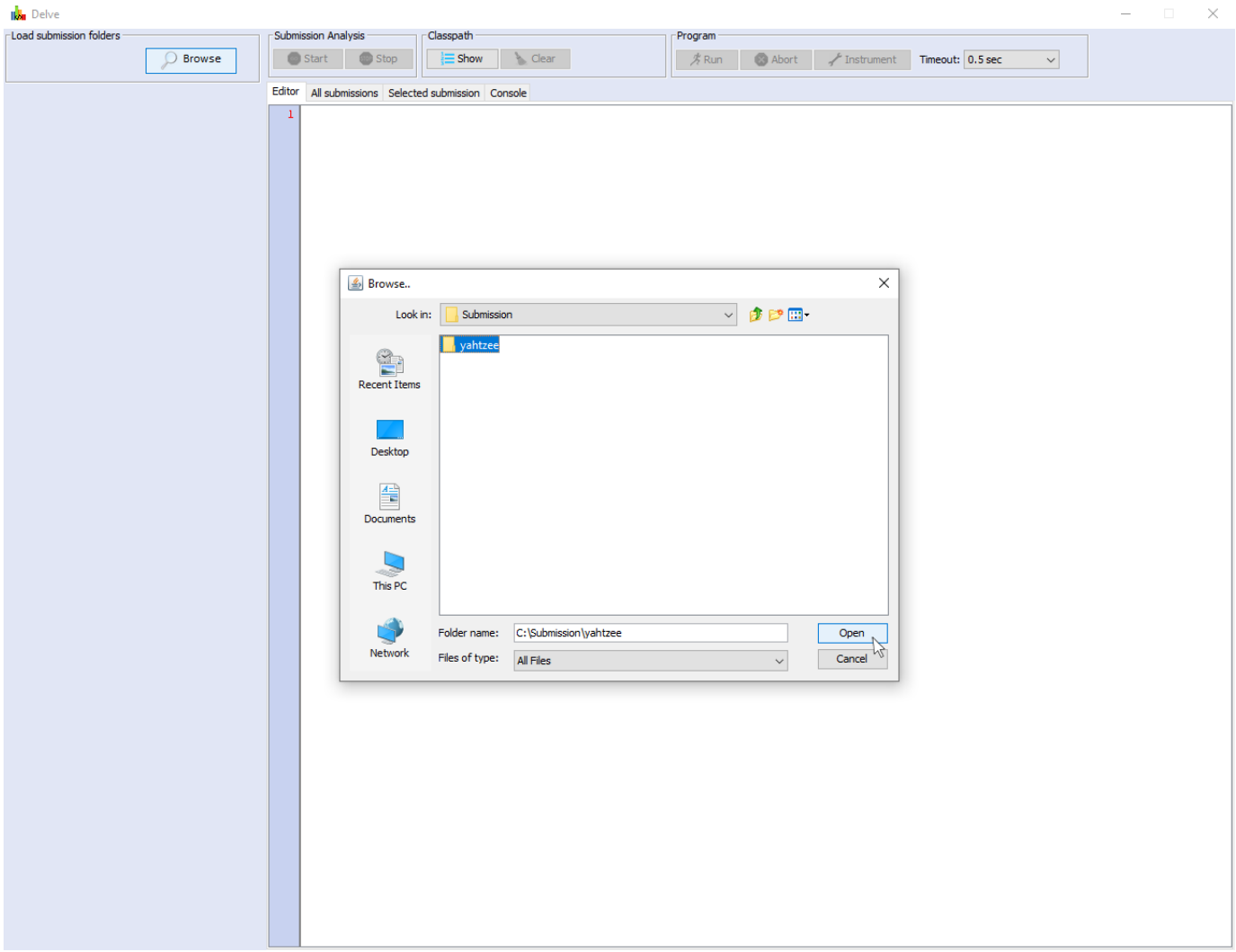

Figure 3: The user can now locate the root directory with the student code snapshots.

At this point, Delve processes the student code snapshots and their timestamps, organizes them into subfolders (one per student) and then analyzes those subfolders to find out how long it took every student to complete the assignment.

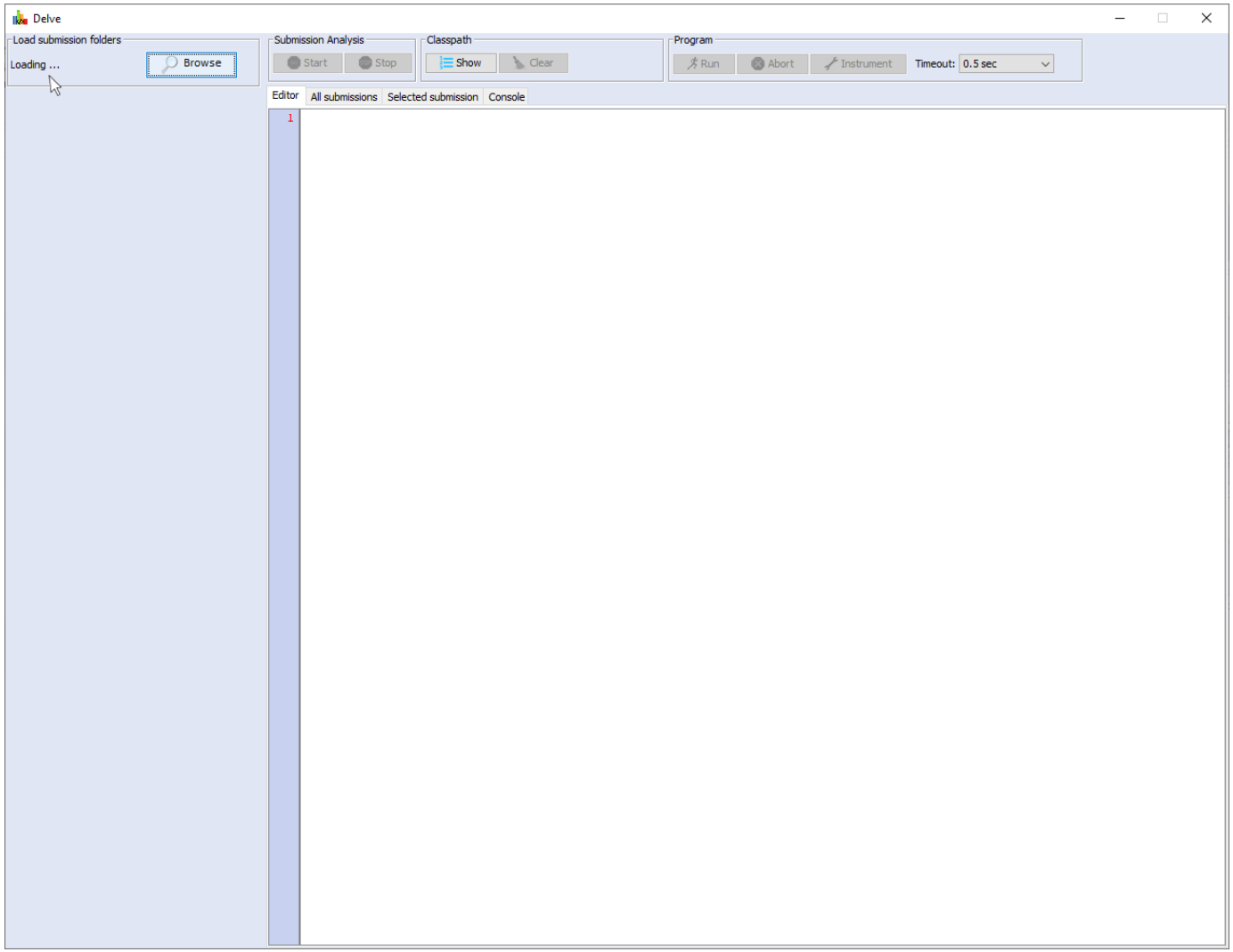

Figure 4: Delve organizes and then processes the student snapshots. This takes a few seconds, up to a minute.

Once done, we can see the student submissions on the left. Delve has created a subfolder for every student, put there the corresponding snapshots and created the file with their timestamps.

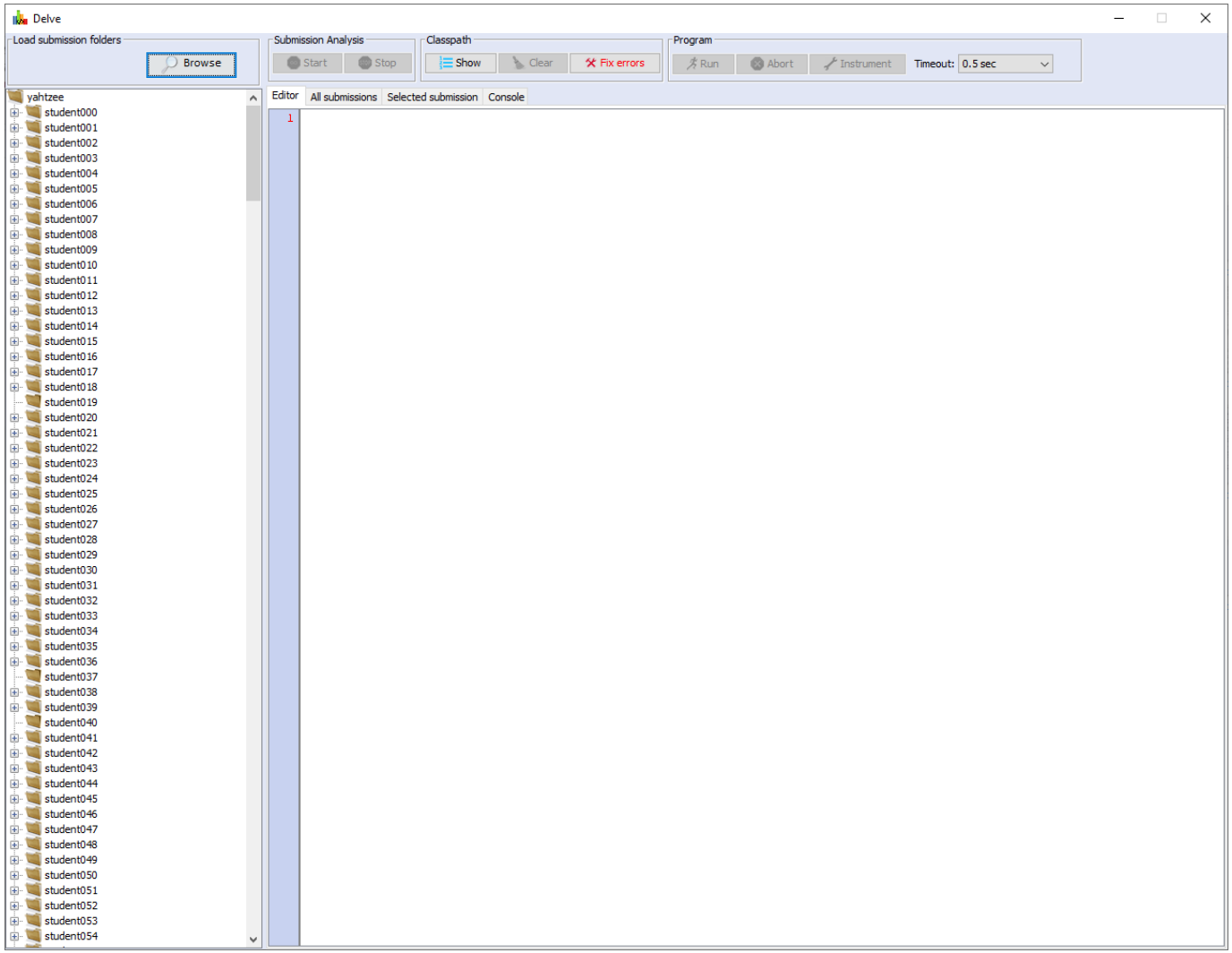

Figure 5: Once the snapshots are loaded they are organized into subfolders and presented on the left.

3. Aggregated Results (classpath is not configured yet)

Tab **`All Submissions`** displays the aggregated results in the form of histograms.

Figure 6: Aggregated histograms (time to complete the assignment, number of lines, methods, ivars and comments).

4. Individual Results (classpath is not configured yet)

The **`Editor`** tab shows a detailed breakdown of the time spent on the assignment by this particular student.

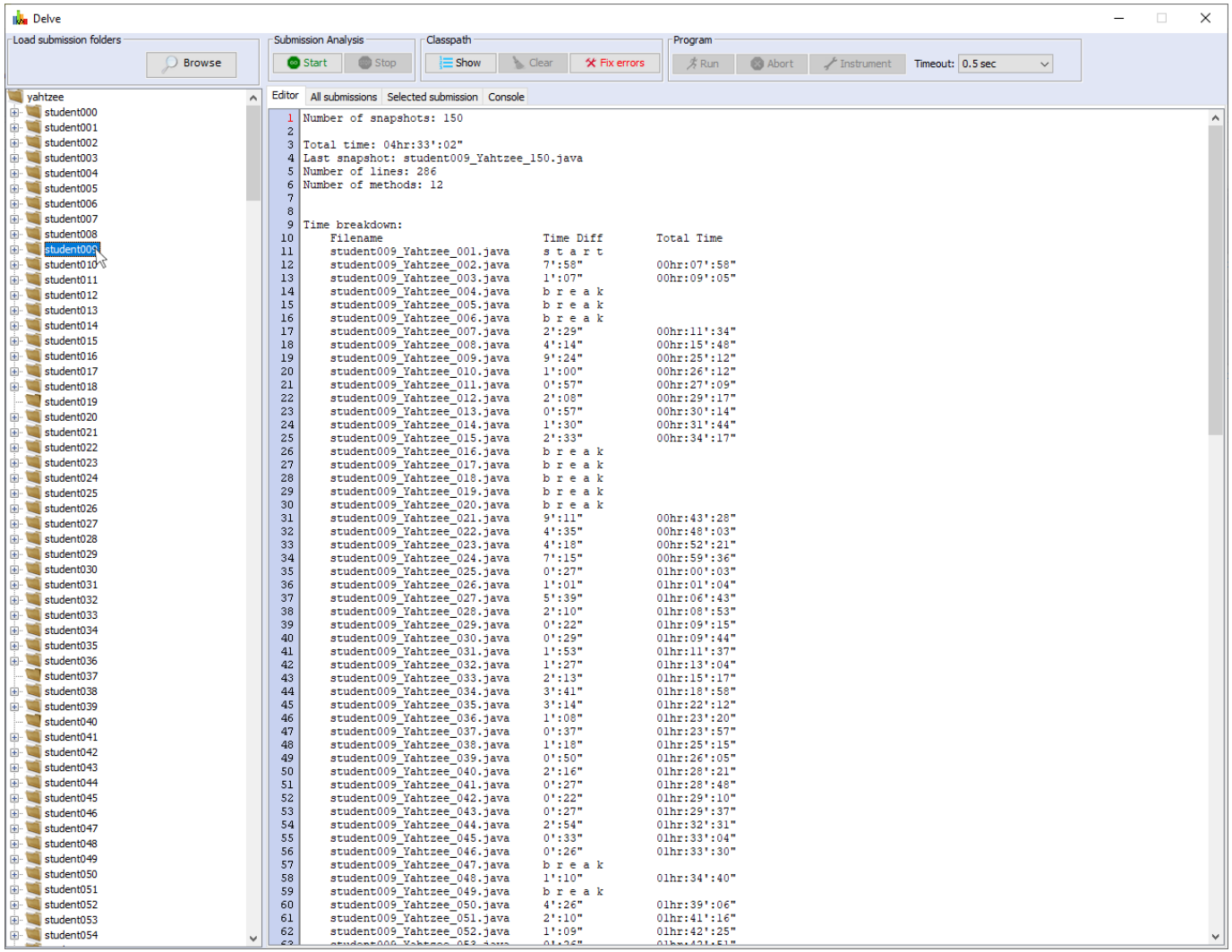

Figure 7:Detailed breakdown of the time spent on the assignment by a particular student.

After pressing ***Start***, Delve analyzes the student snapshots and display how several features progress over time.

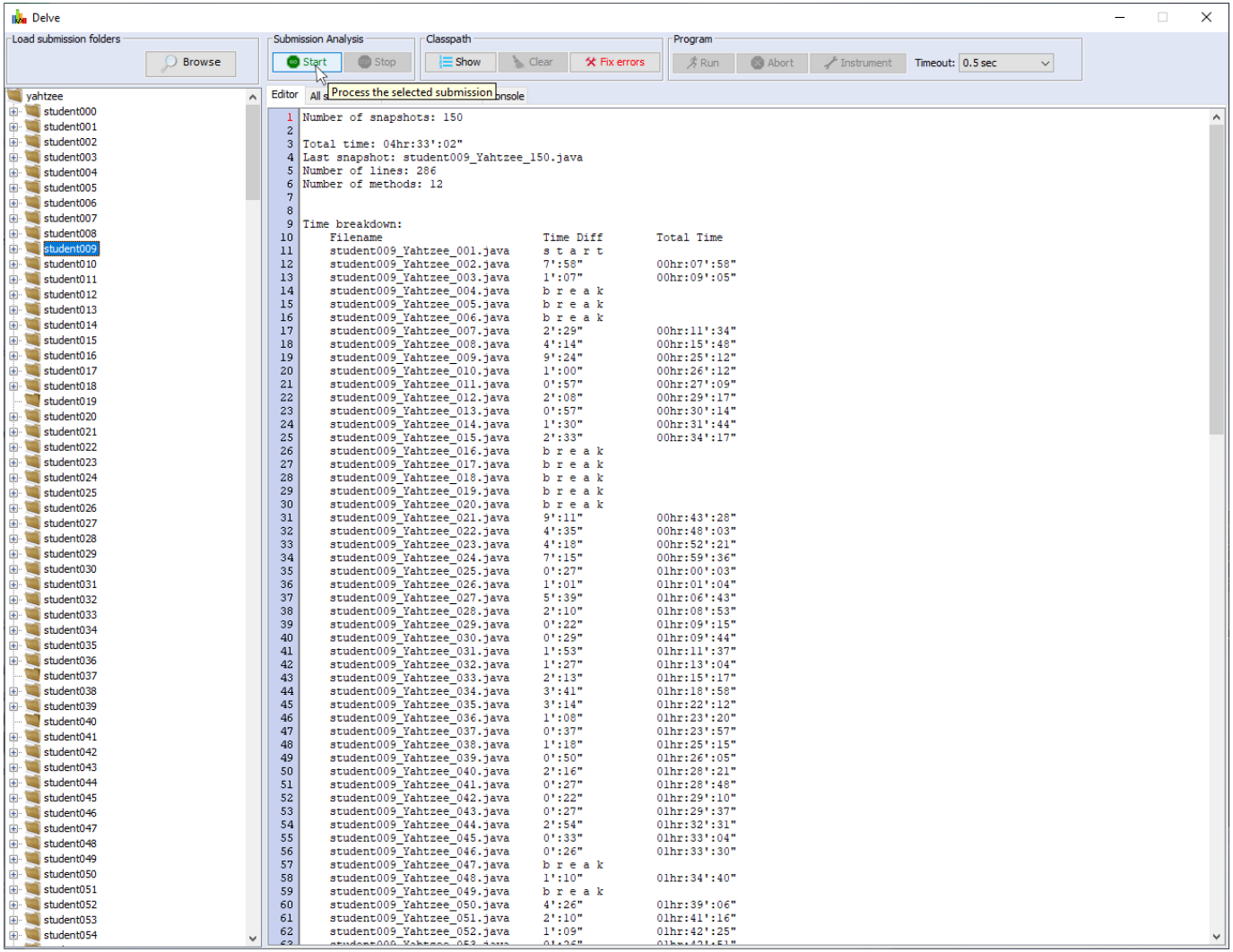

Figure 8: As soon as the user clicks on the *Start* button, Delve begins analyzing the selected submission.

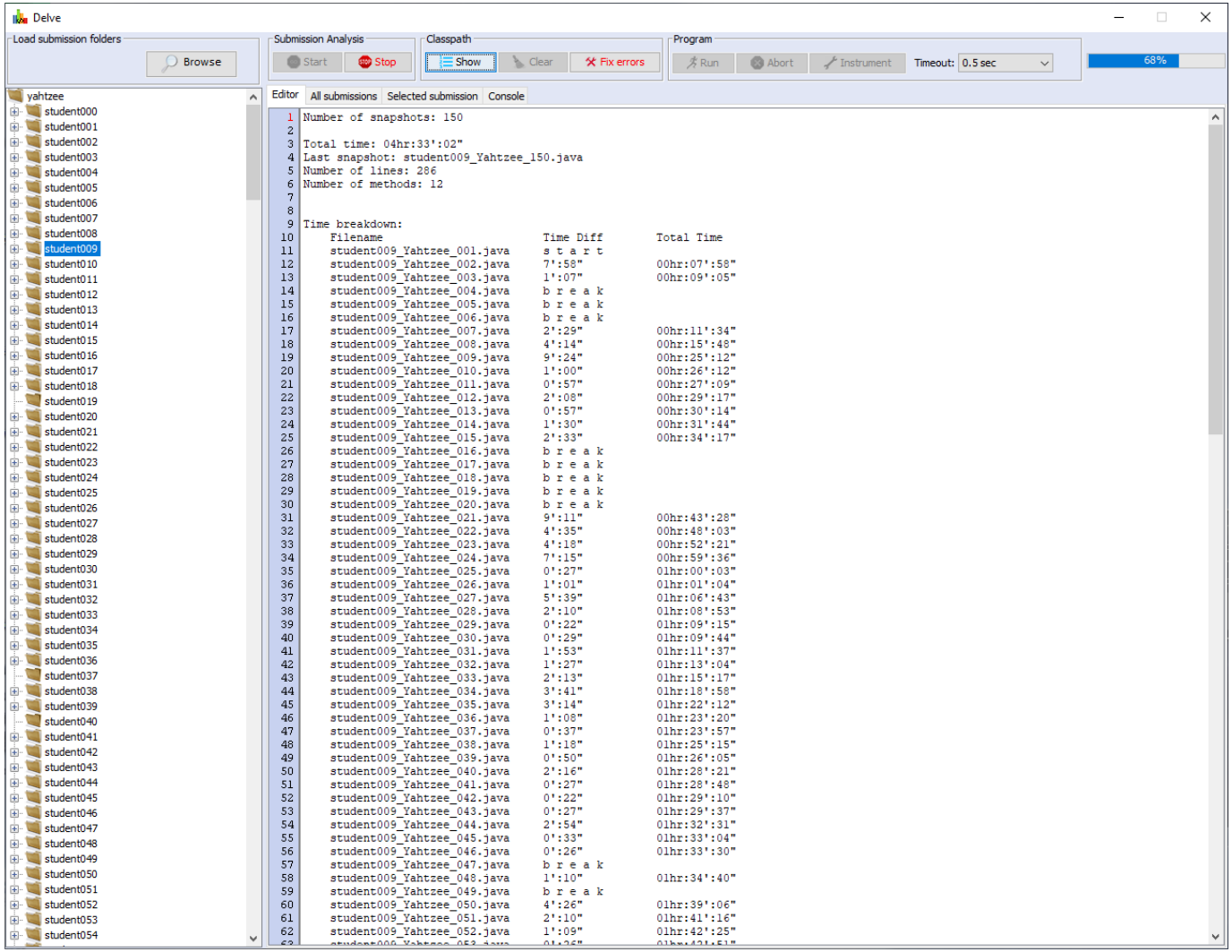

Figure 9: Delve analyzing the selected submission.

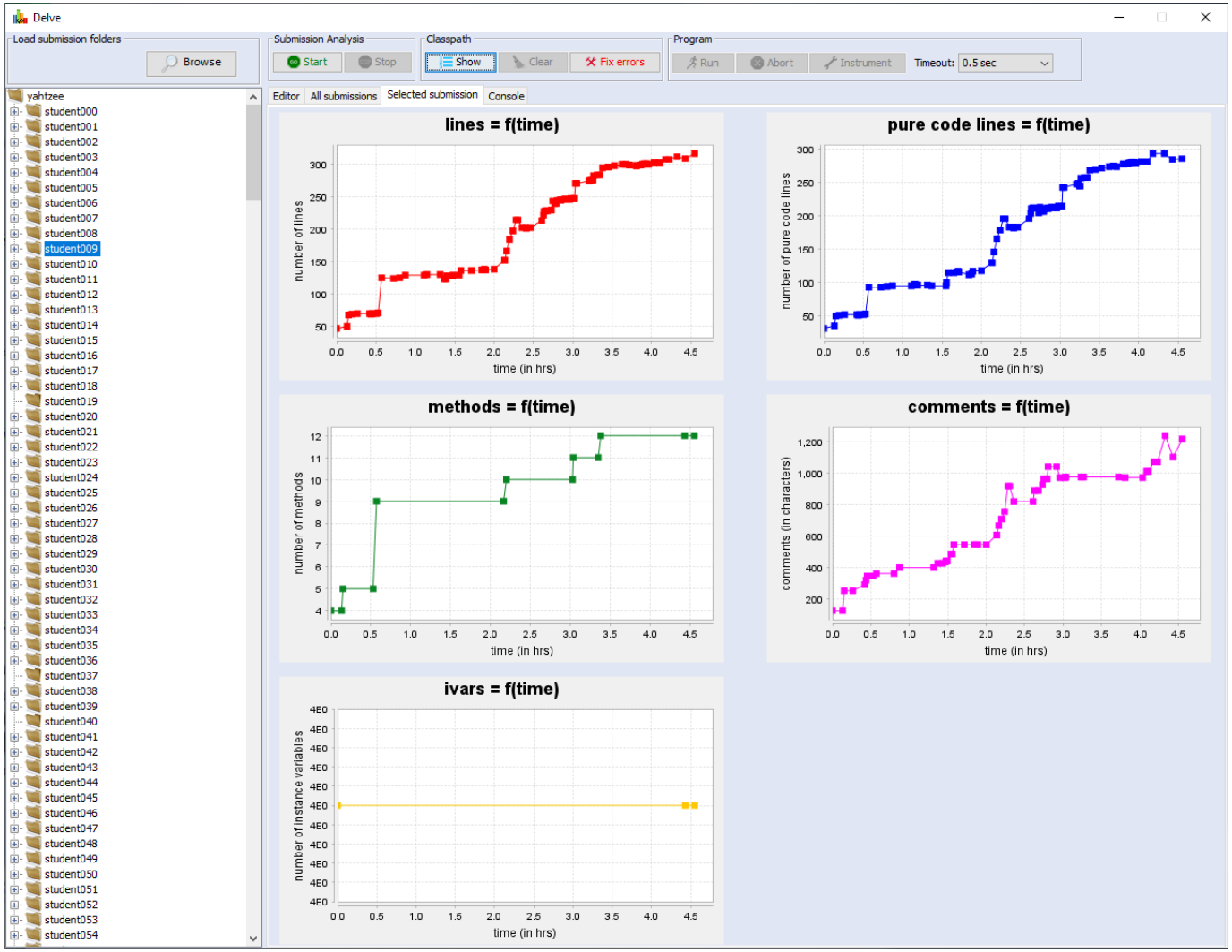

Figure 10: Time plots of various features for an individual student. Note that all plots have the same number of points (measured values for the same time), only points that changed since the last measurement are displayed to avoid overloading the plots. For example, the number of lines kept changing slightly from 0.5hrs to 2hrs, whearas the number of methods was constant for the entire period (although we took the exact same number of points).

5. Program Execution

The user can expand the folder with the student submission and select anyone of its snapshots.

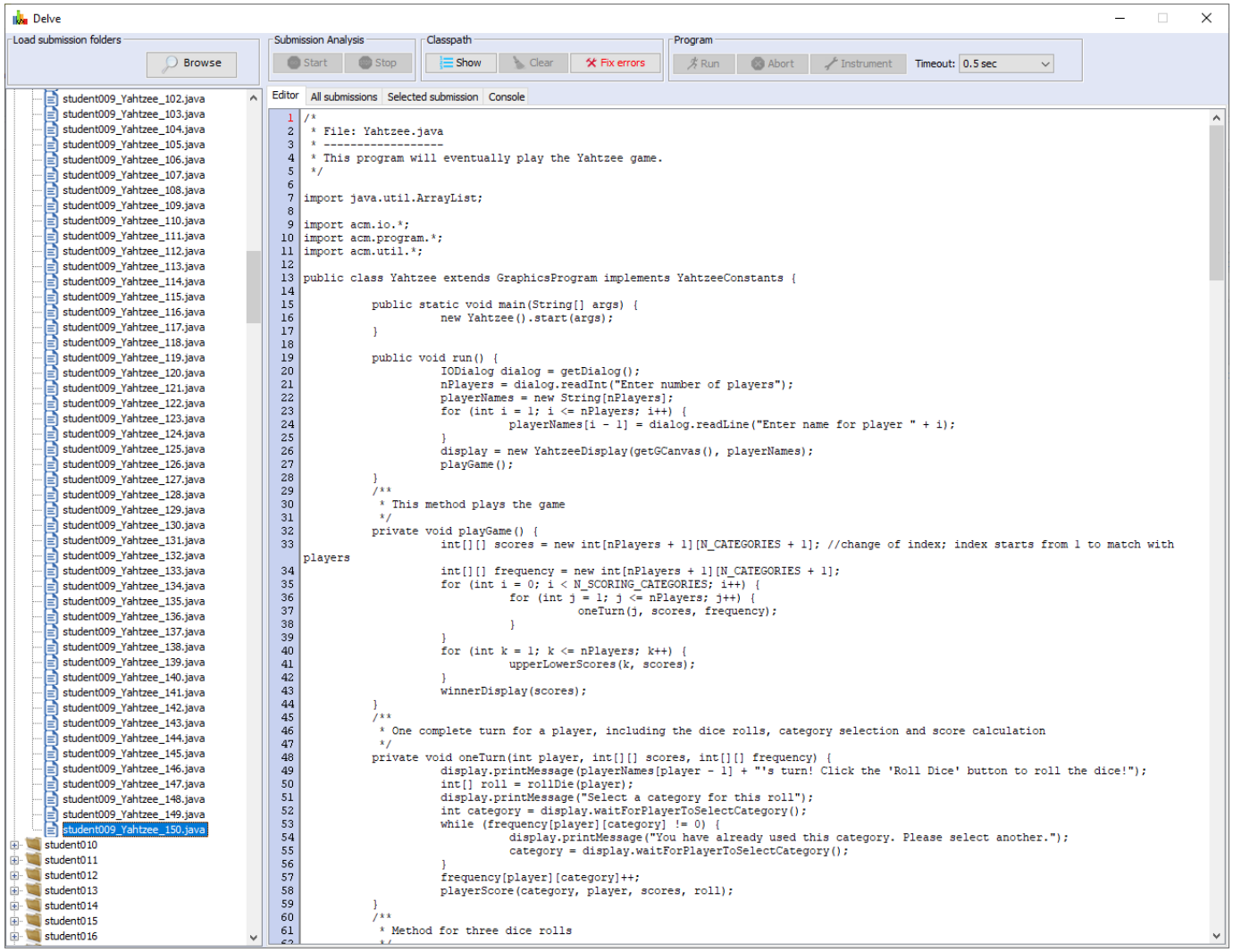

Figure 11: As soon as we selecting a snapshot, the source code appears in the editor.

We notice that there are one or more files missing from the classpath and therefore the code does not compile successfully.

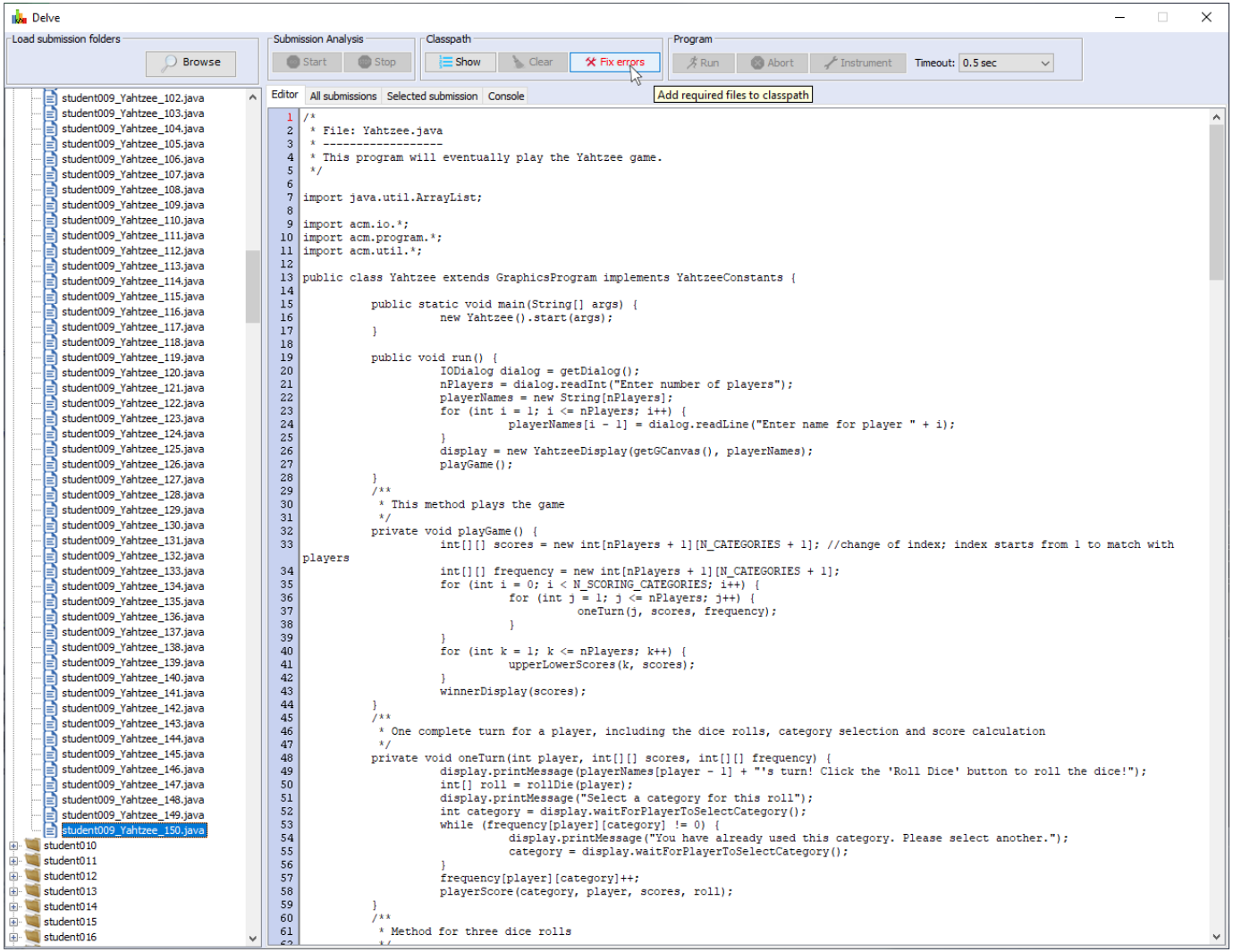

Figure 12: Fixing the classpath is needed to run the program.

Typically, those are the starter files that accompany the assignment (e.g. libraries or other files). In the example, the assignment has three starter files (two jar files and a file with constants).

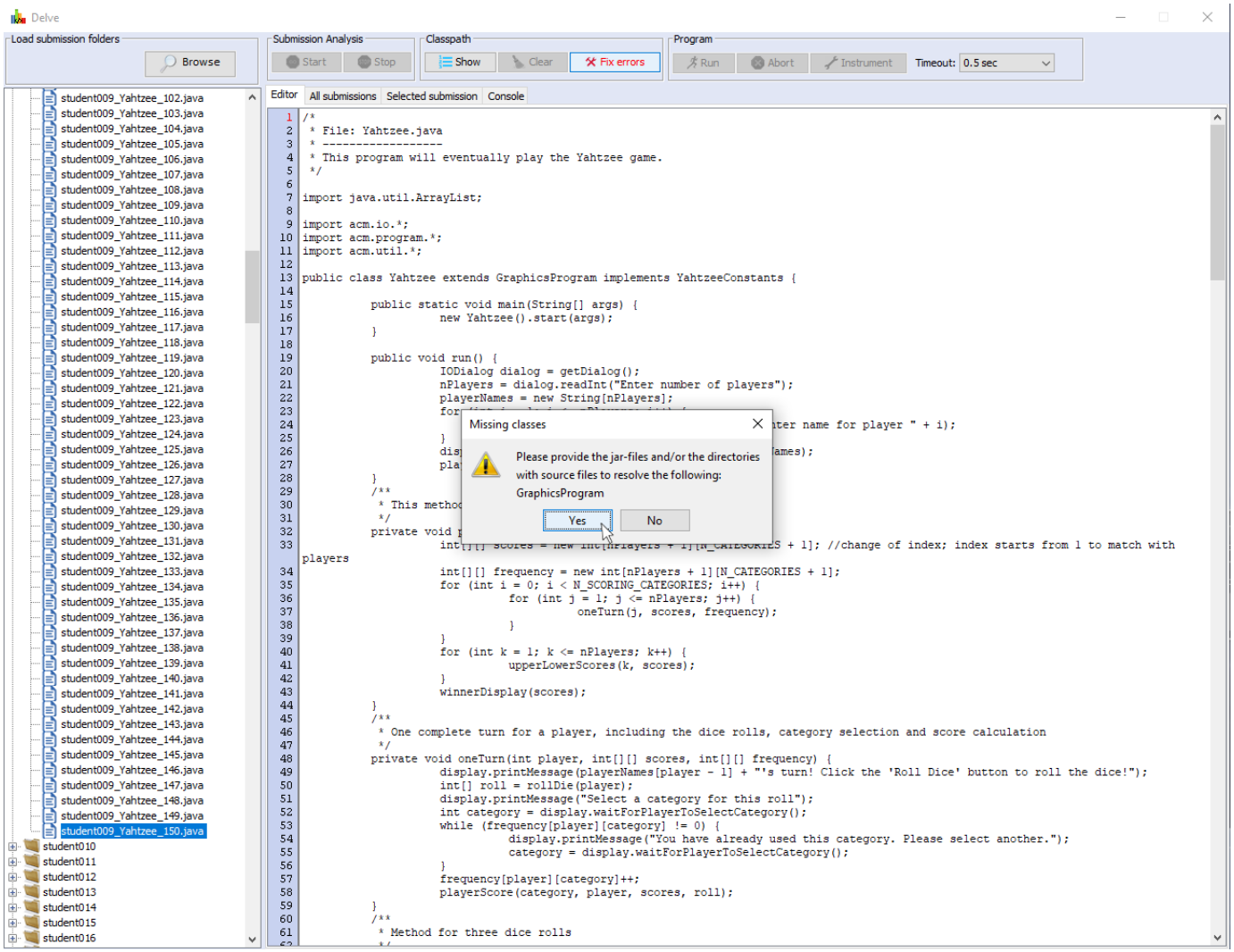

Figure 13:Preparing to add the first file that is missing from the classpath

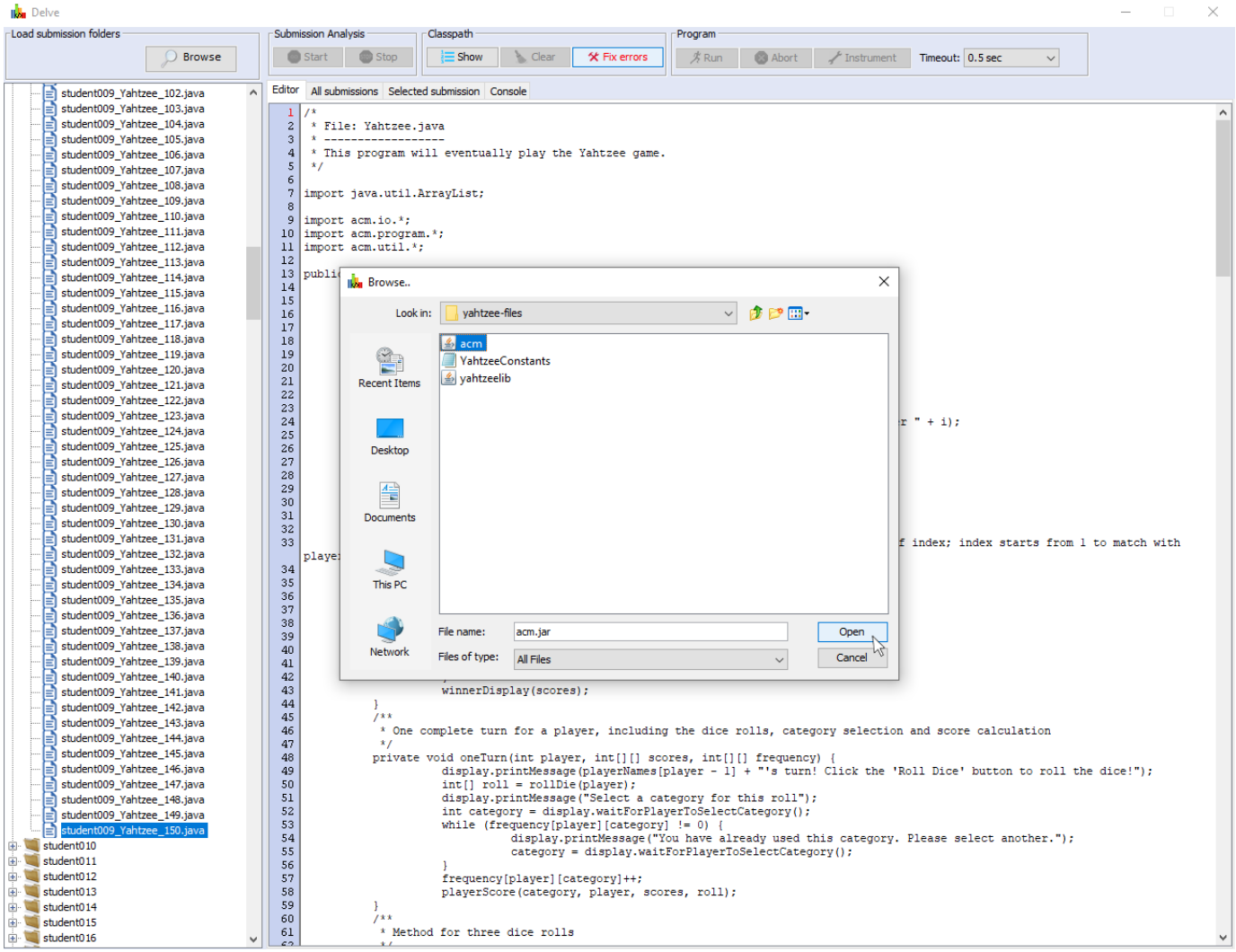

Figure 14: Selecting the first of the three files that are missing from the classpath.

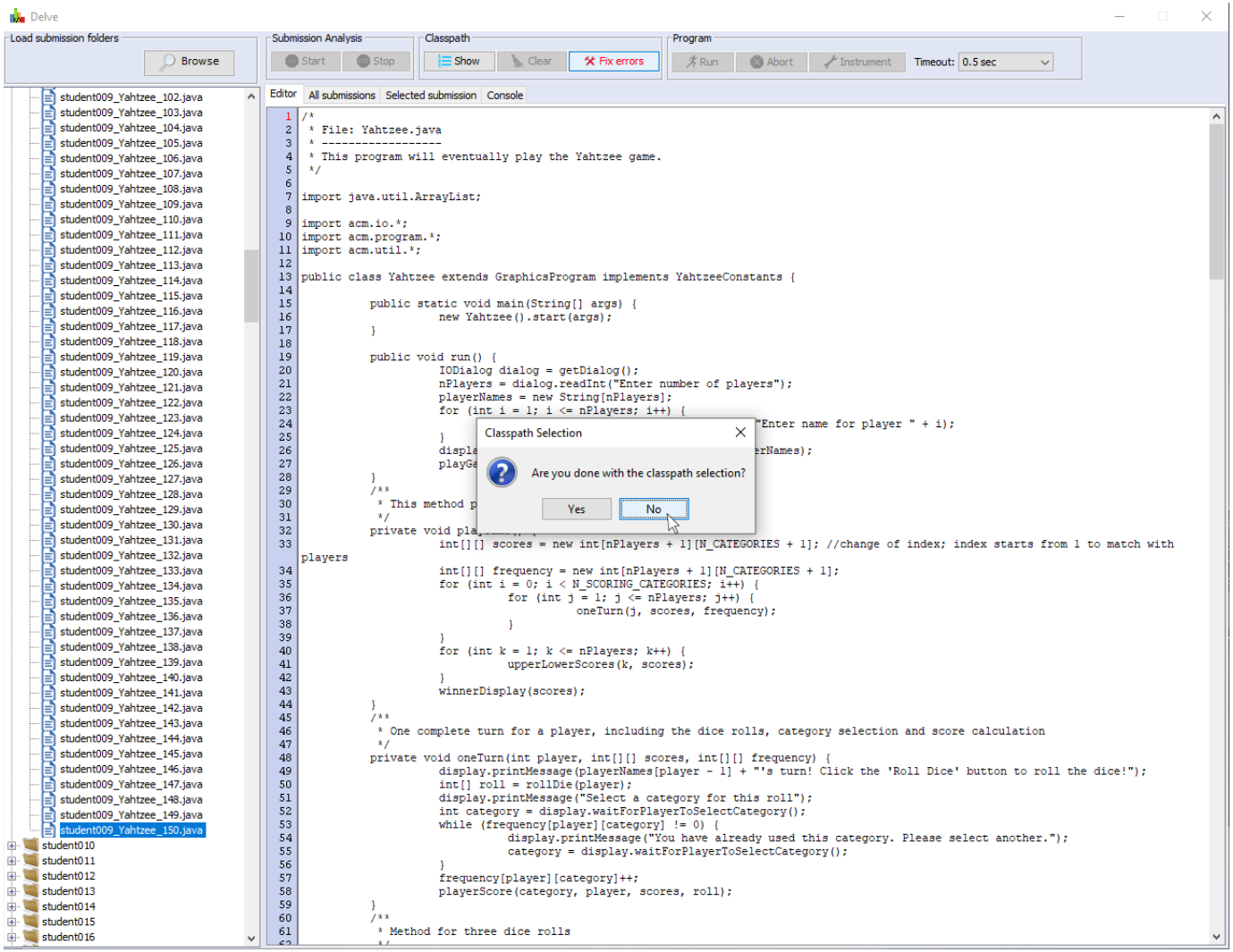

Figure 15: There are more files missing from the classpath. Preparing to add the second required file to the classpath.

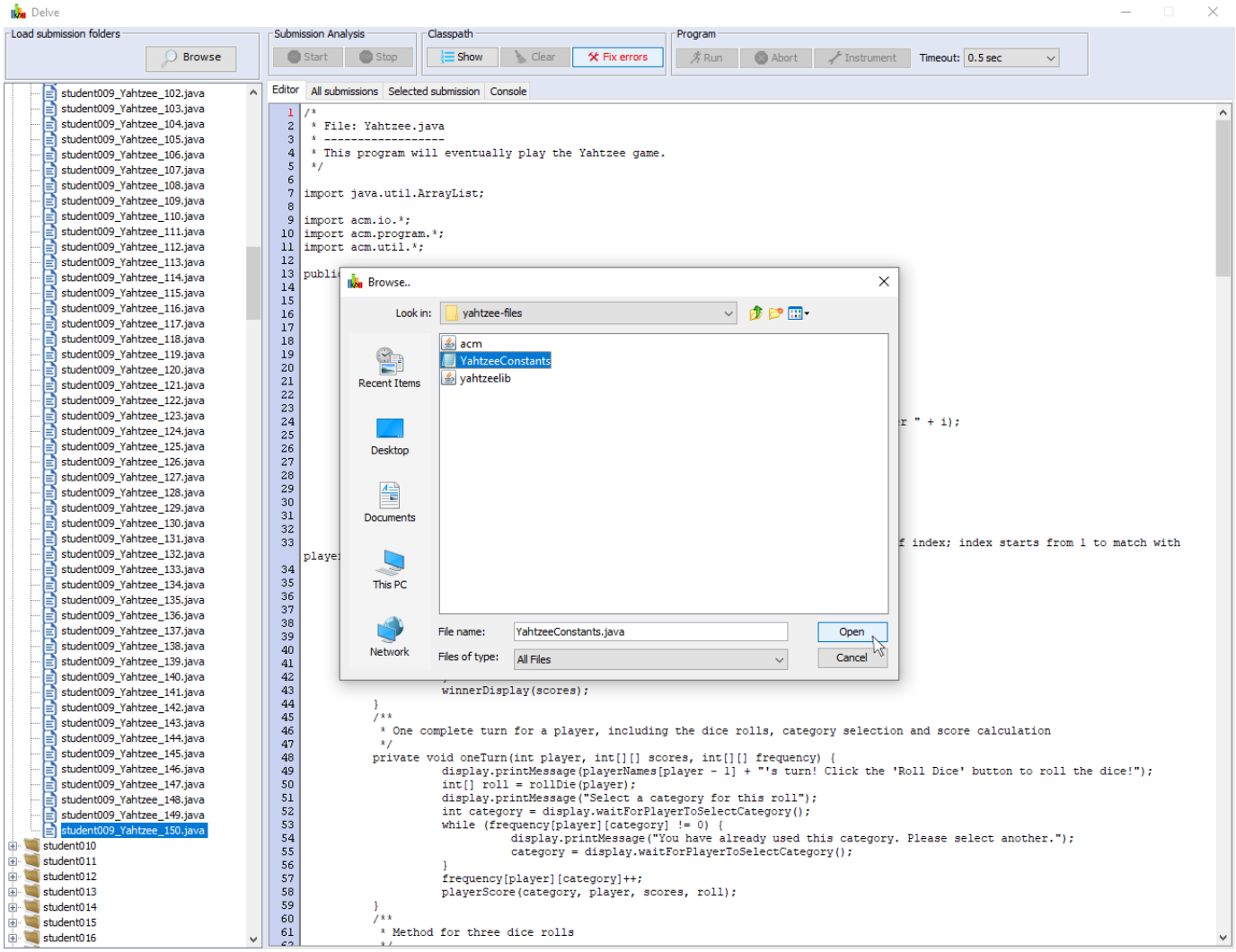

Figure 16: Selecting the second of the three files that are missing from the classpath.

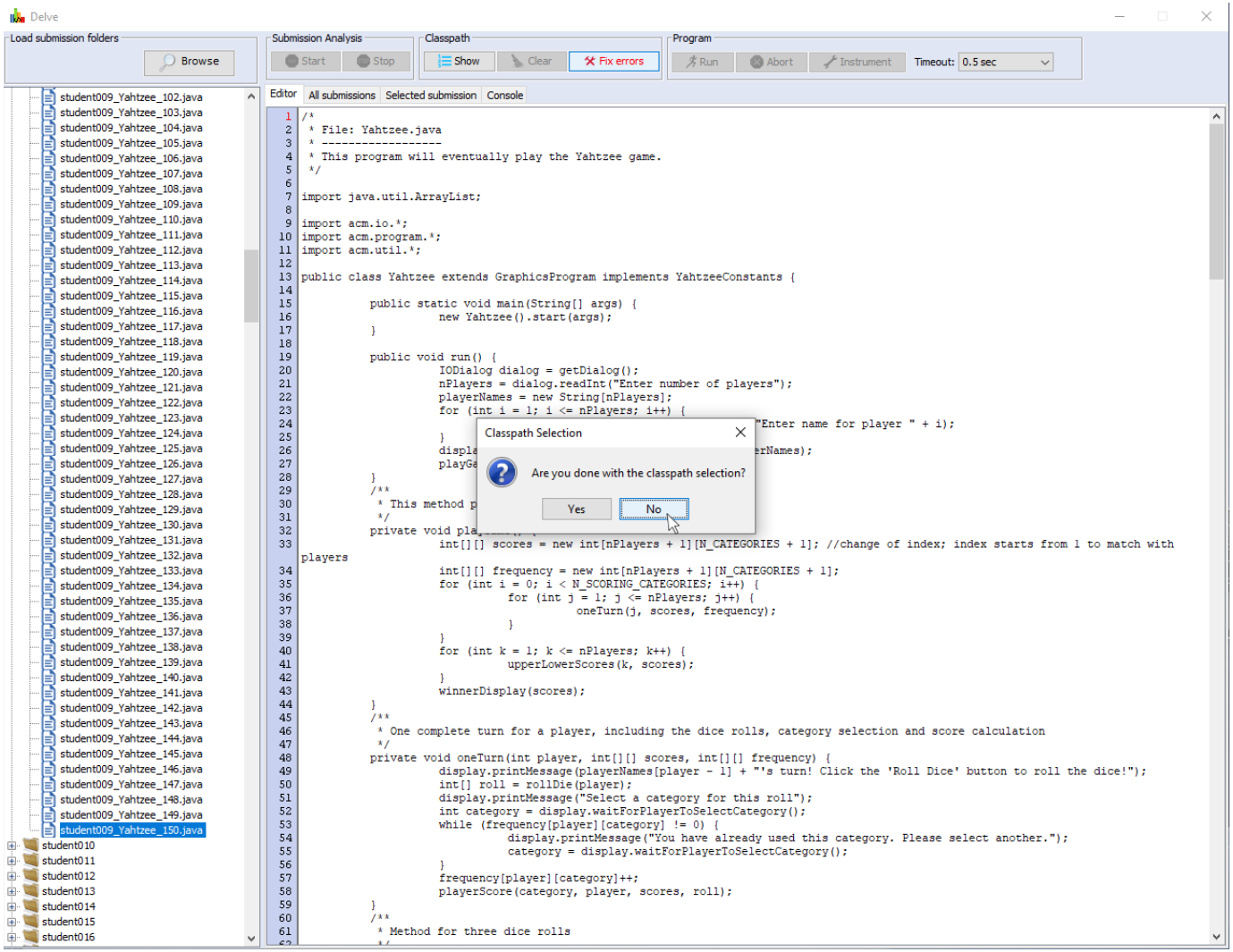

Figure 17: There is one more file missing from the classpath. Preparing to add it.

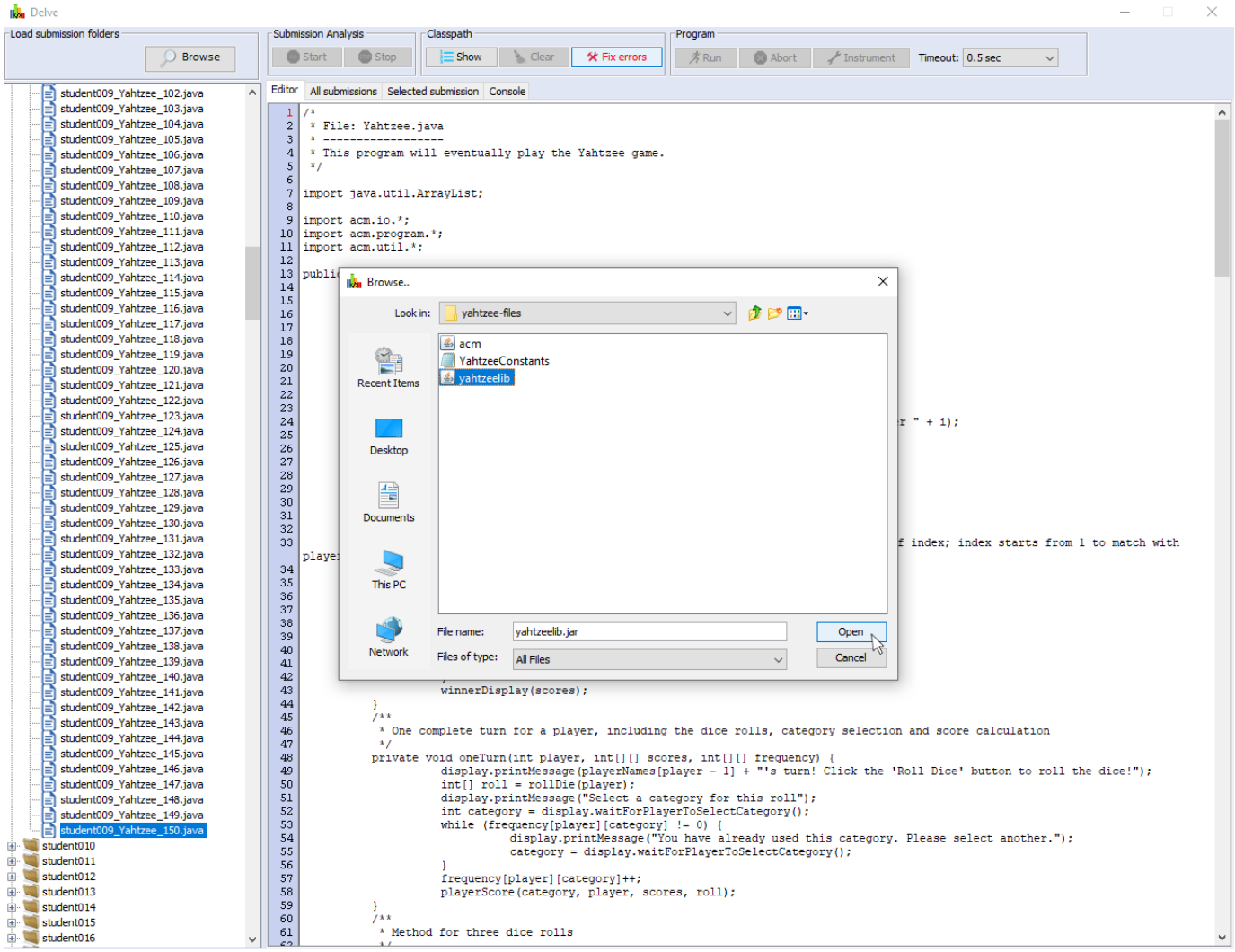

Figure 18: Selecting the last of the three files that are missing from the classpath.

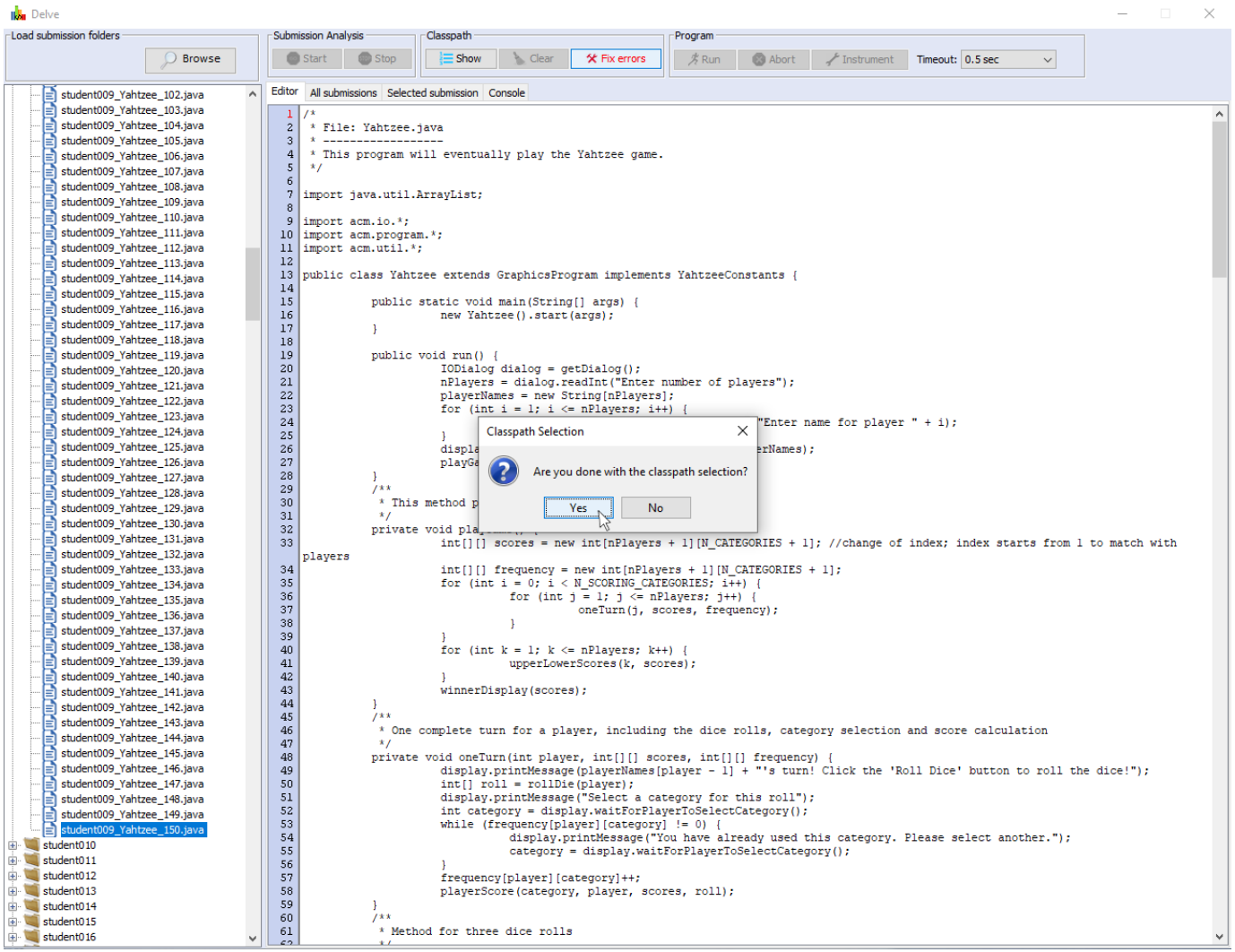

Figure 19: There are no more files missing from the classpath.

Once we have added one or more files, Delve tries to determine if the classpath is correctly configured. It does that by analyzing the final snapshots for every student as those are expected to compile. If 75%+ of them compile, the classpath is configured successfully.

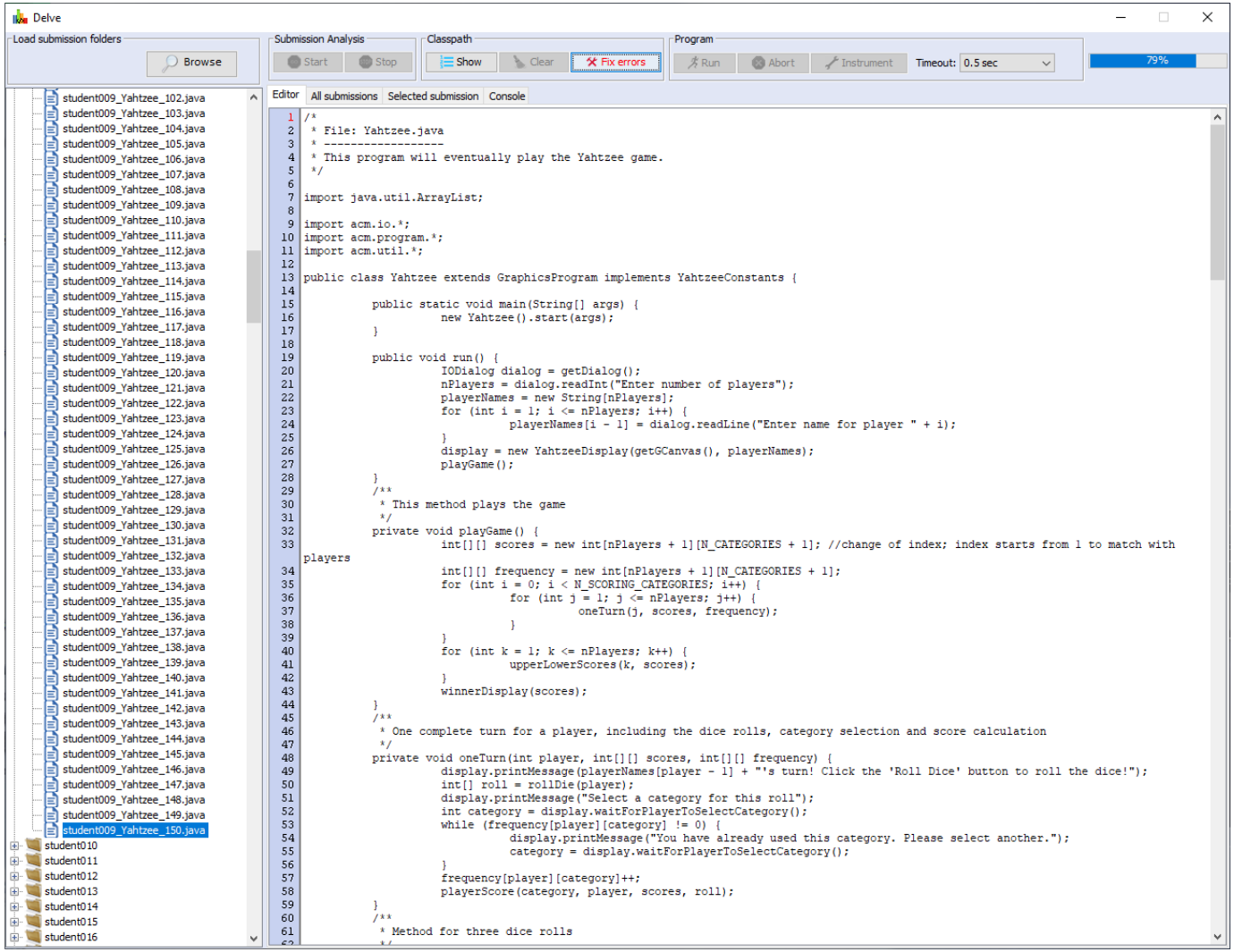

Figure 20: Delve tries to determine if the classpath is correctly configured by trying to compile the last snapshots of every student.

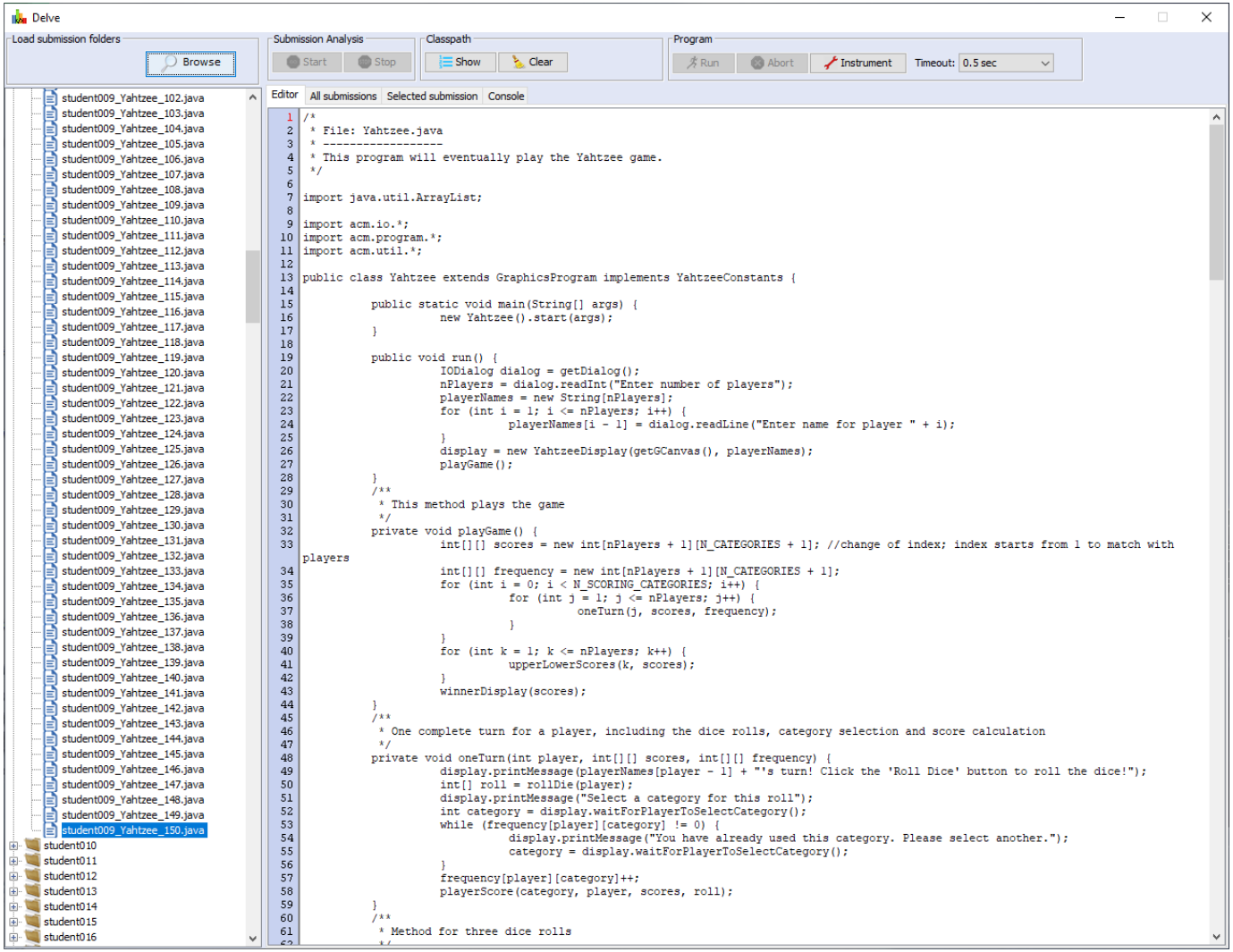

Figure 21: The button *Fix errors* disappeared which means that the classpath is configured correctly.

At any point, we can check the classpath.

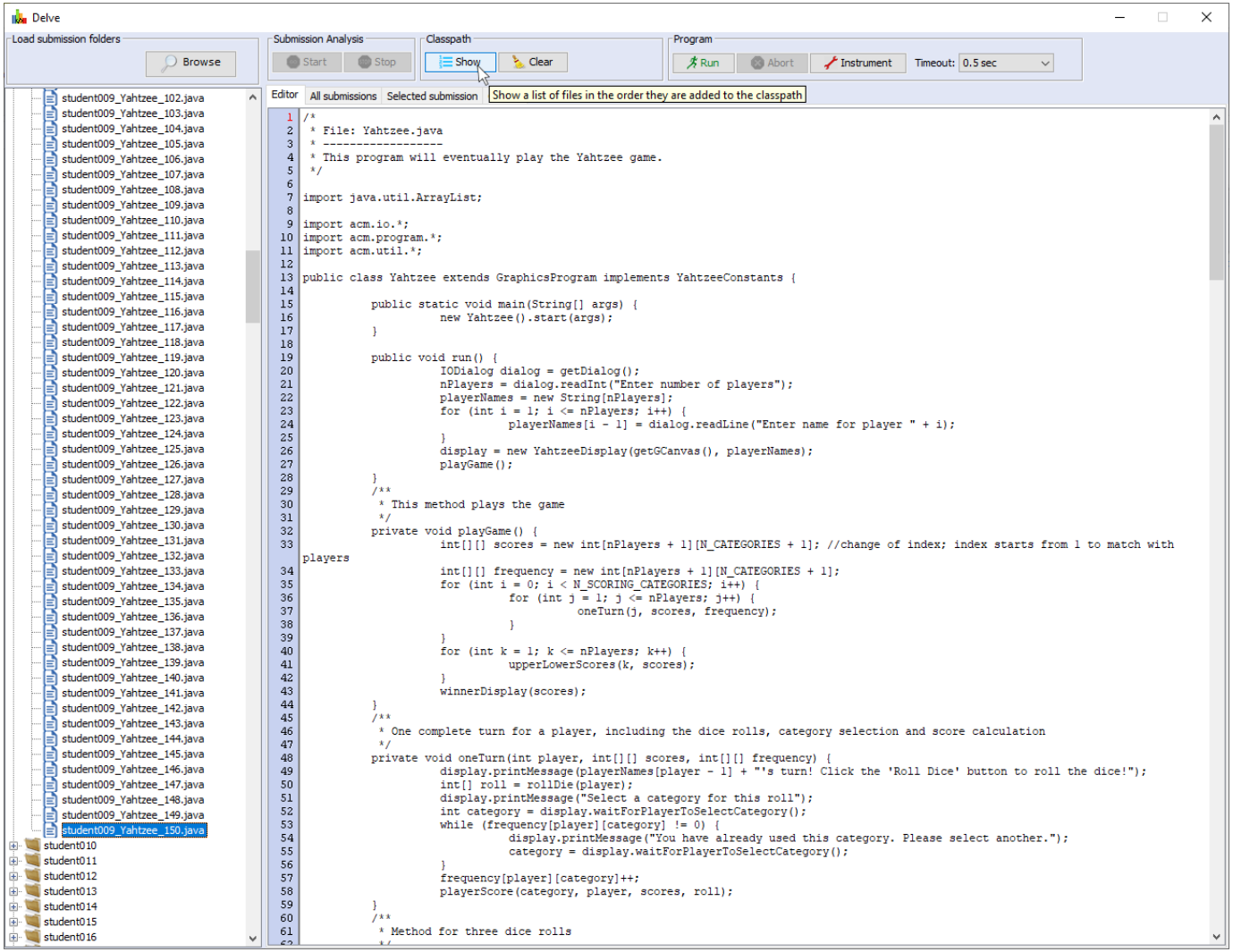

Figure 22: We can clock button 'Show' at any point in time to see the existing classpath.

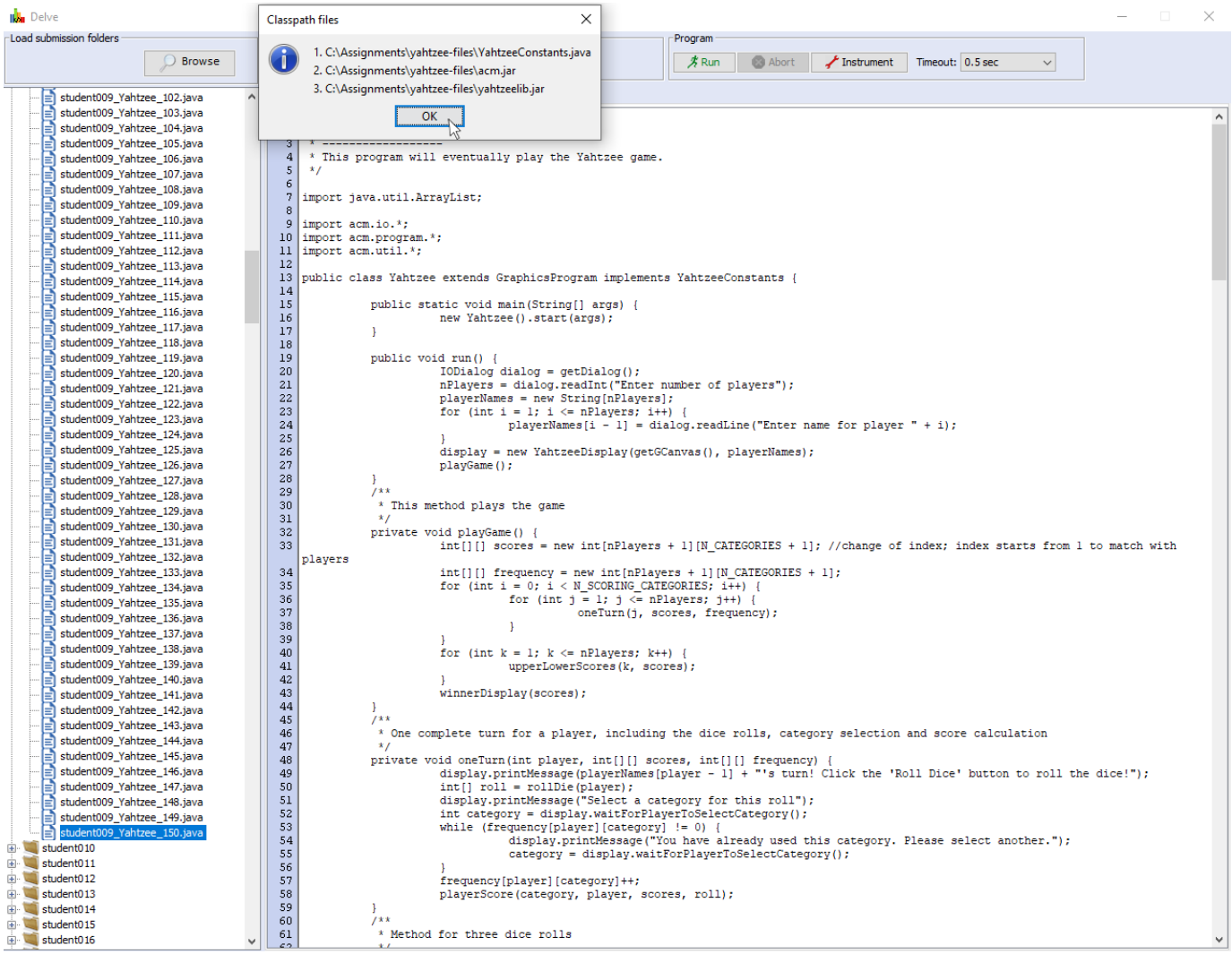

Figure 23: A small pop-up window displayes the three files that we added to the classpath.

Now, every snapshot that compiles and includes method **`public static void main(String[])`** can be executed. When an executable snapshot is selected button **Run** is enabled.

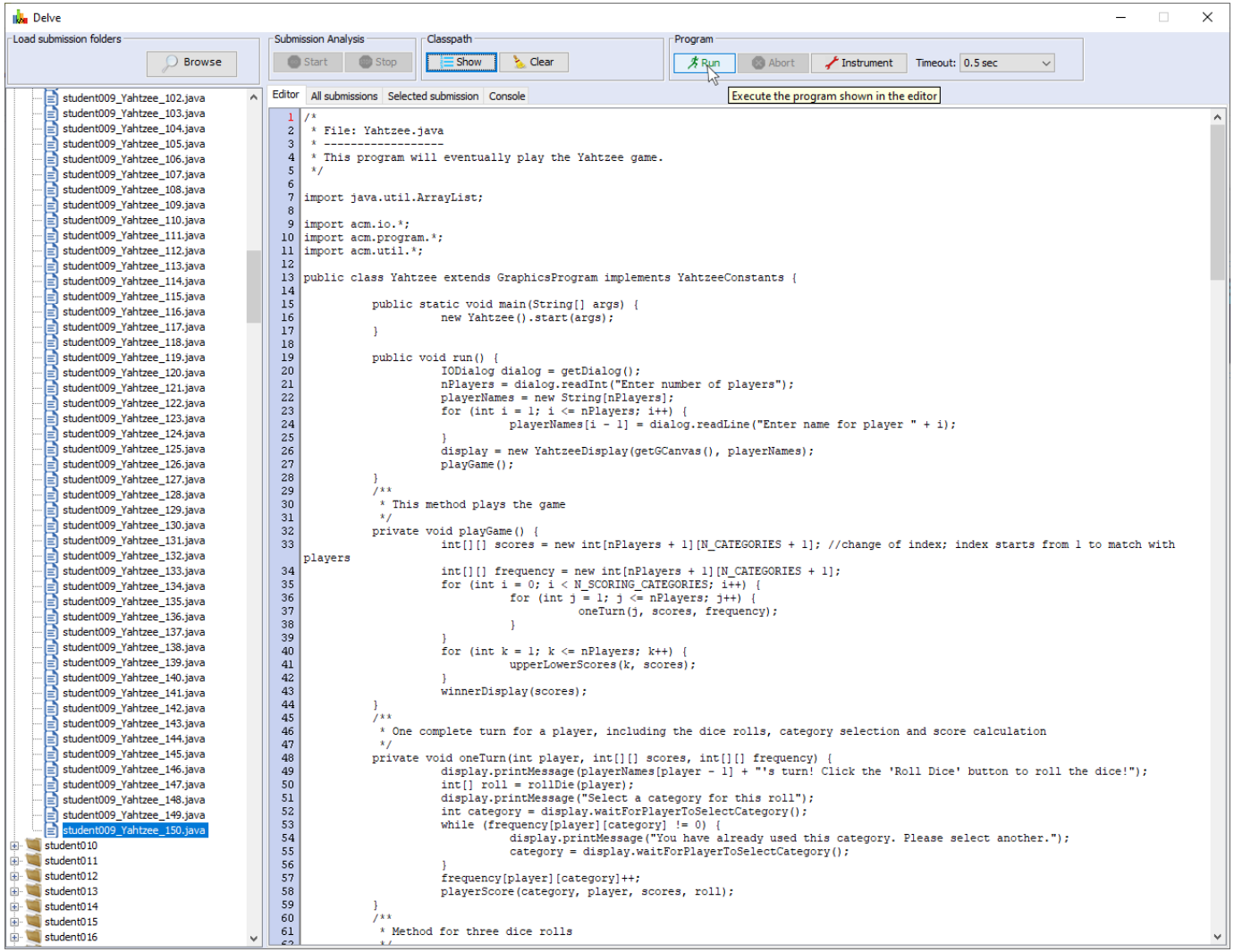

Figure 24: Snapshot *student009_Yahtzee_150.java* compiles and includes also *public static void main(String[])*. Therefore, button *Run* is enabled, ready to execute the program.

Pressing **Run** kicks off the program execution.

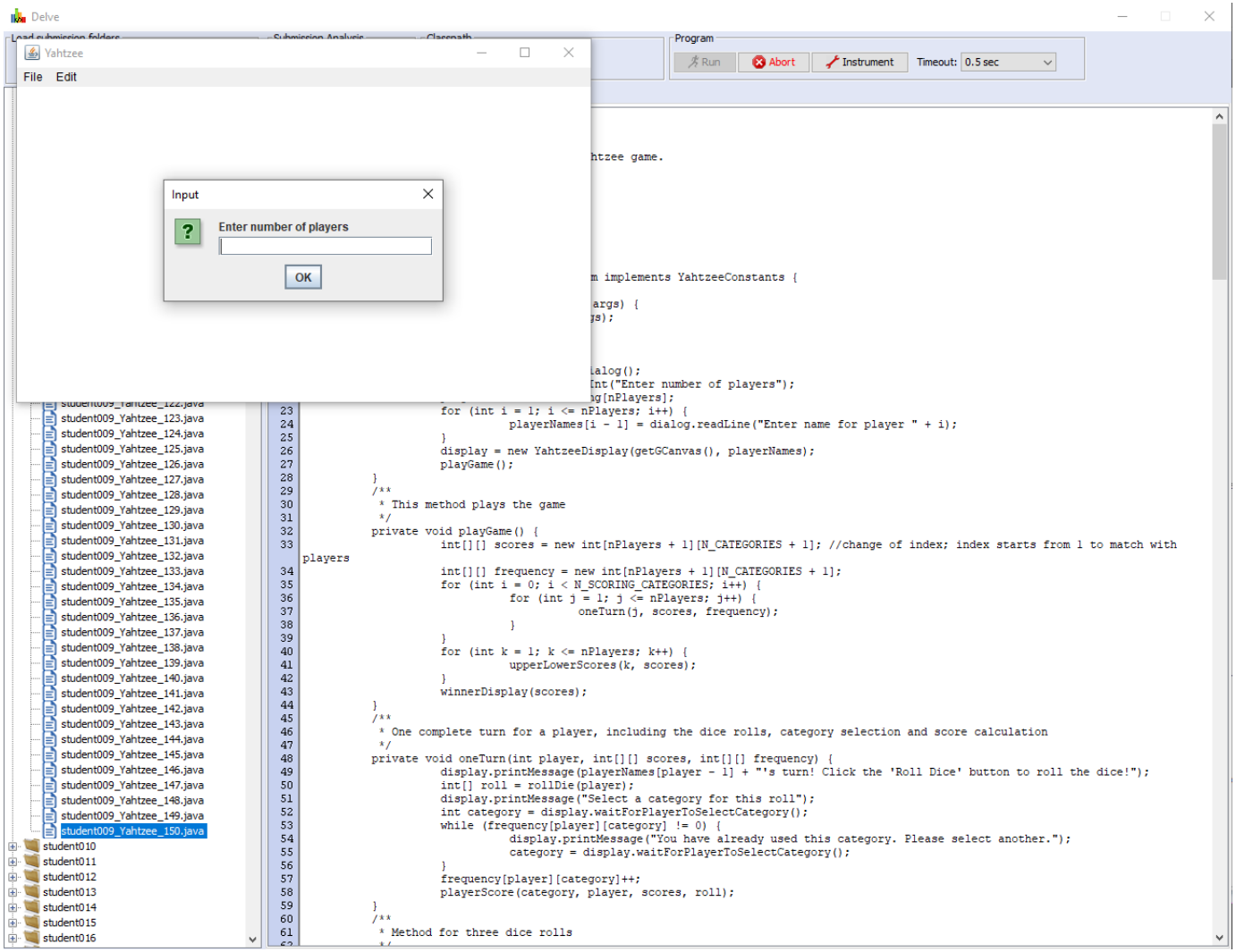

Figure 25: To execute the program we simply press *Run*.

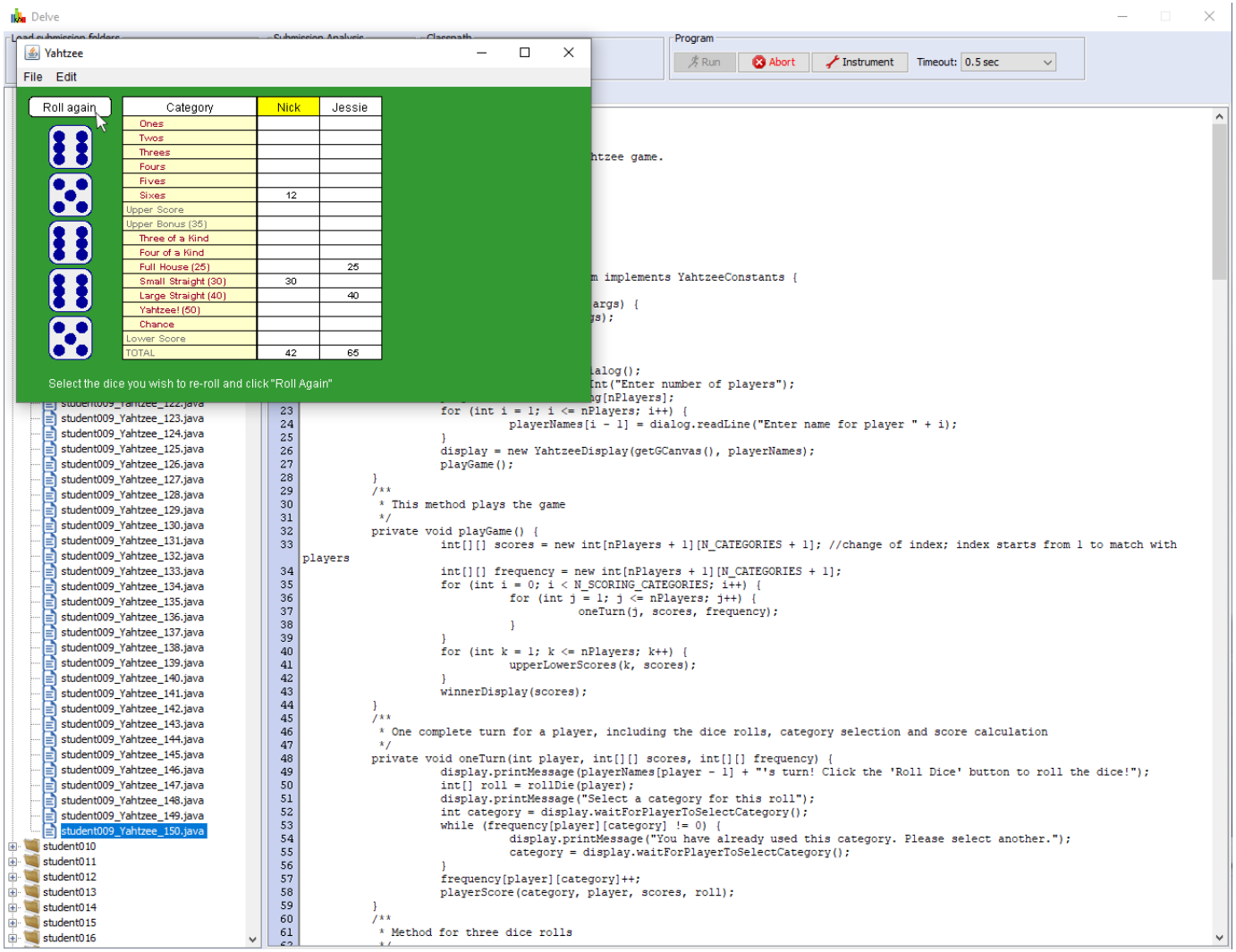

Figure 26: The program runs normally, just like in an IDE environment.

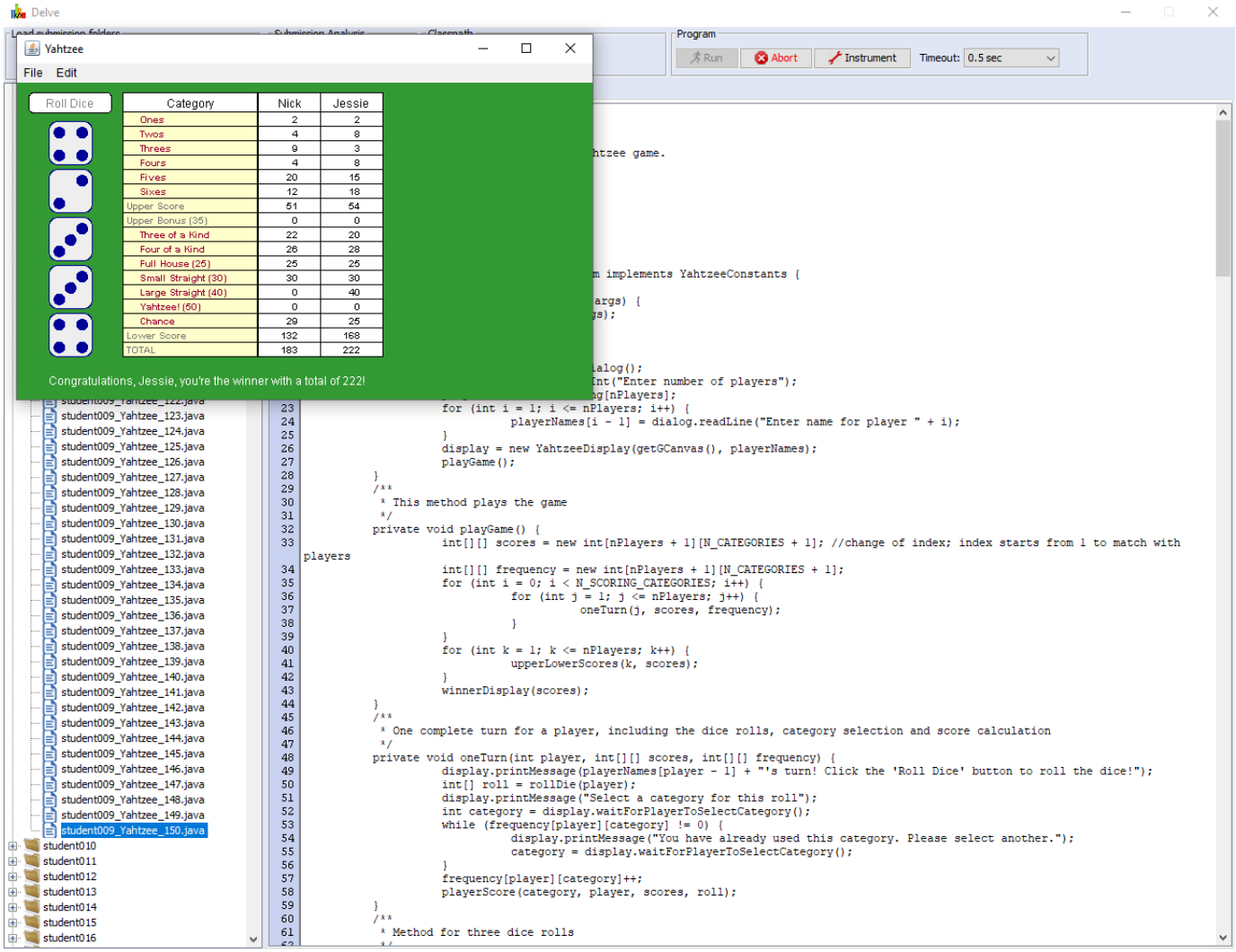

Figure 27: The program runs eventually to completion.

6. Individual Results (classpath is configured)

Now that classpath is configured, we can press **`Start`** to get additional analysis results about an individual submission.

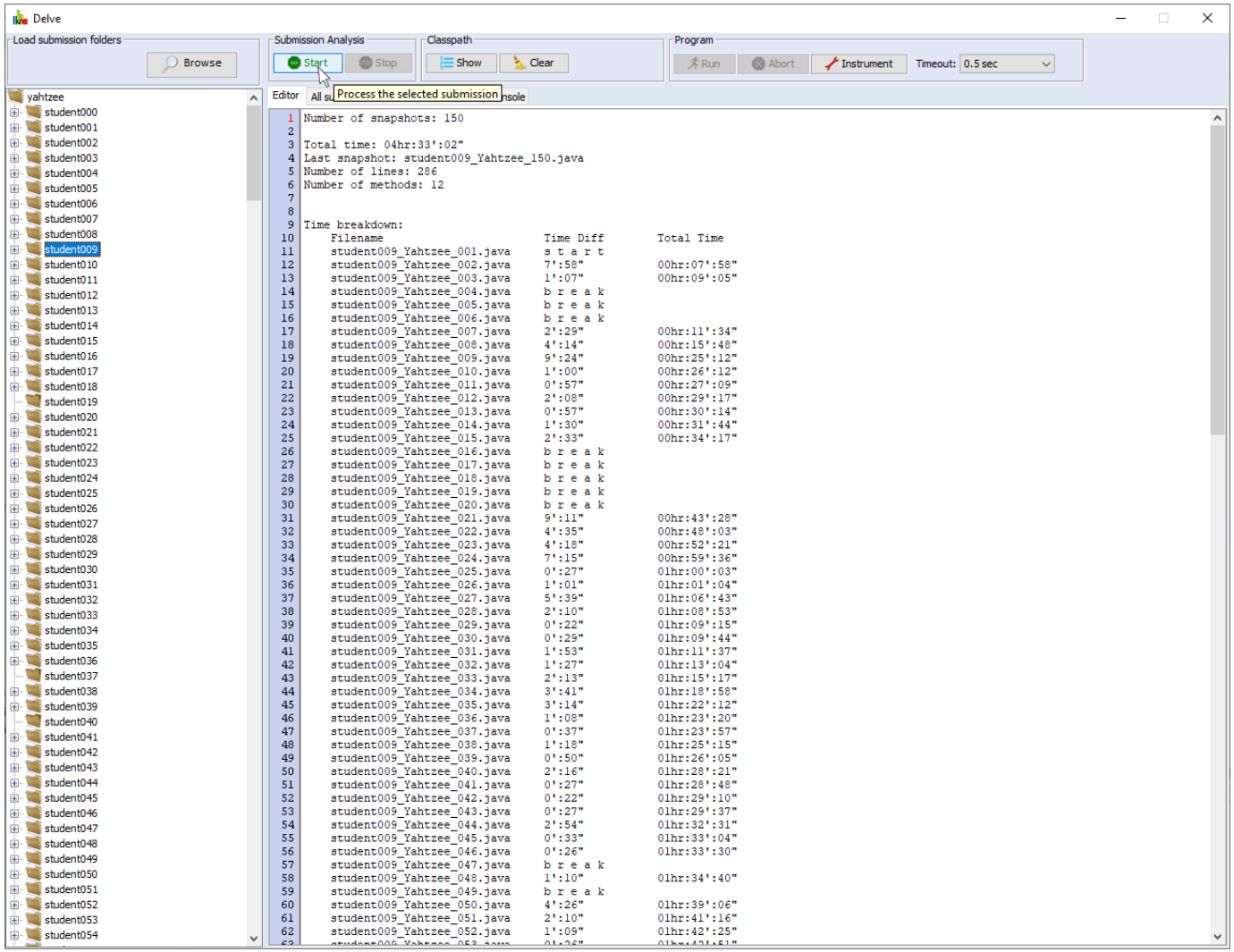

Figure 28: On *Start*, Delve begins analyzing the submission to get additional results now that classpath is configured.

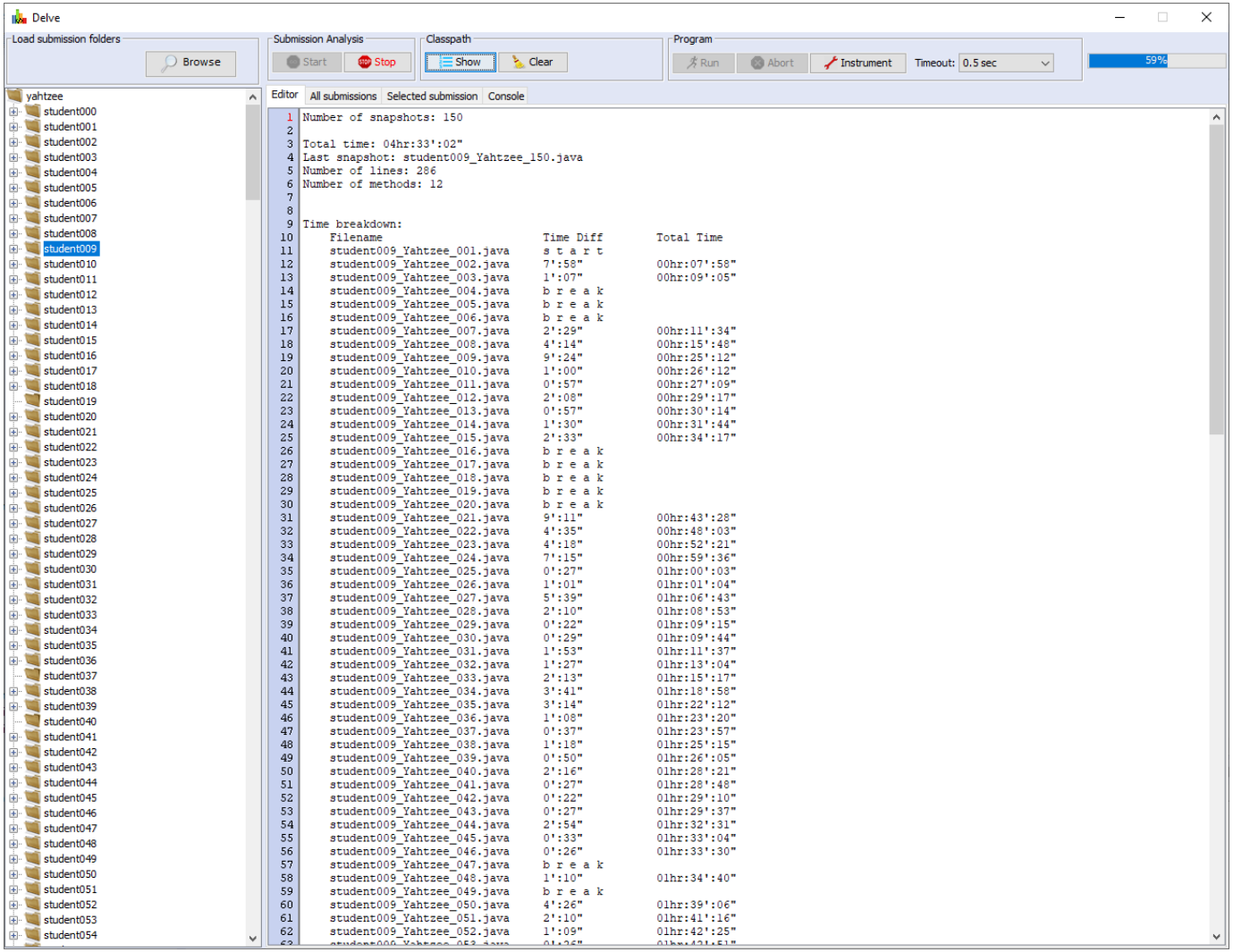

Figure 29: Analyzing an individual submission takes a few seconds.

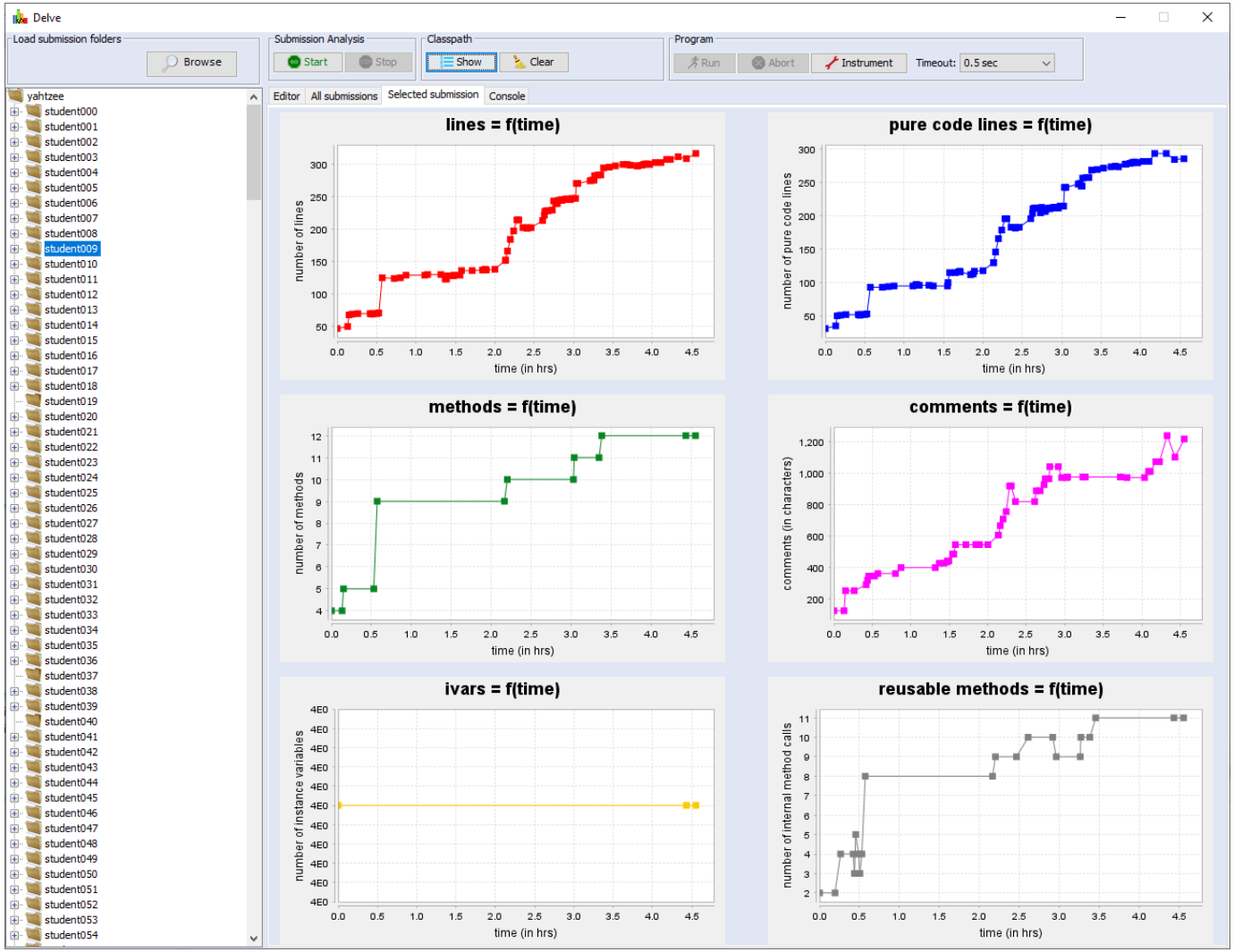

Figure 30: Compared to before setting the classpath (***Figure 10***), we notice that Delve displays the number of calls to user-defined methods which is an indicator of code reusability.

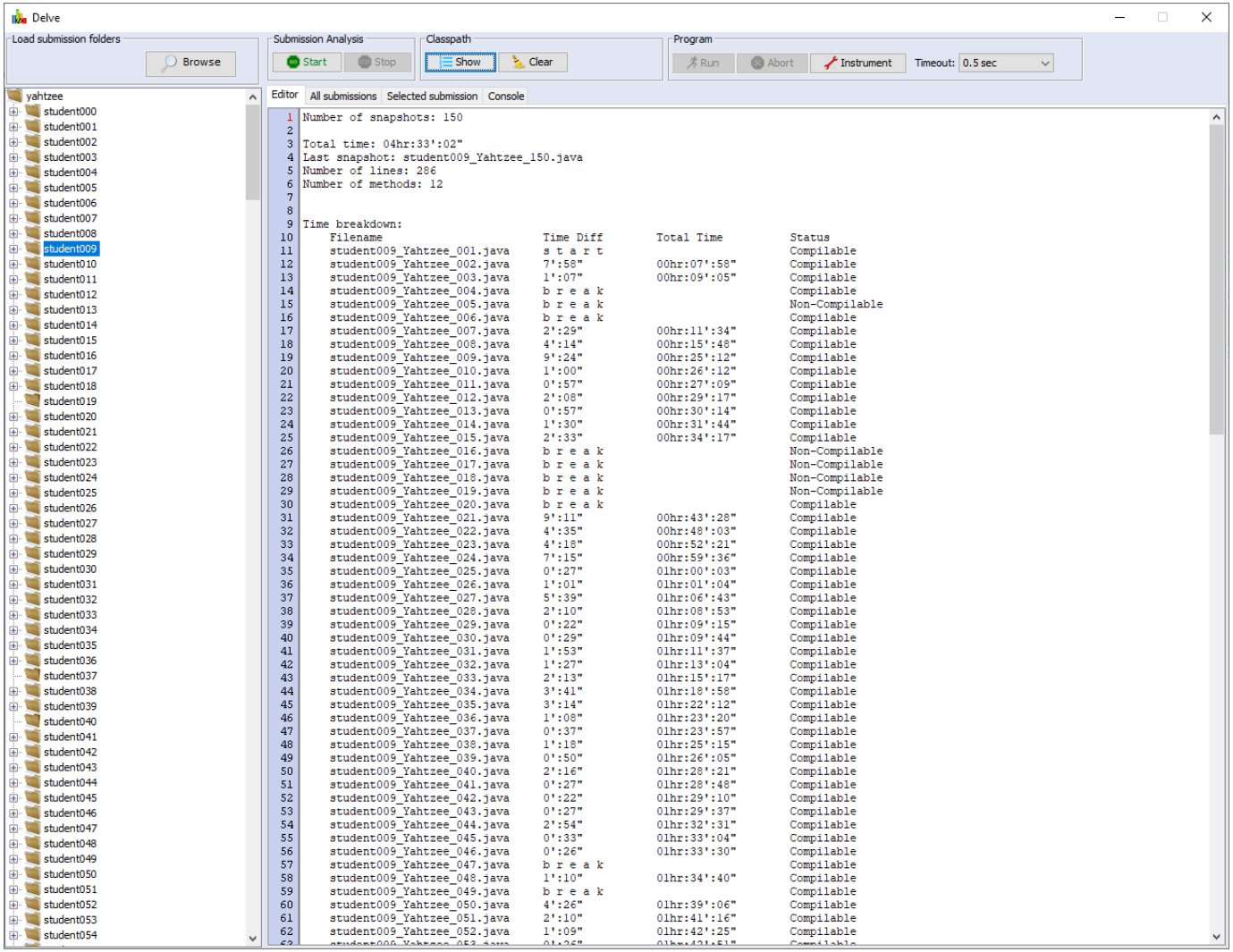

Figure 31: The *Editor* tab displays an extra column for the status of each snapshot.

7. Code Instrumentation

Delve facilitates testing the program functionality at scale, by assisting the user instrument the code to save the program state. The idea is that if a program is **`deterministic`** and we can capture its state over time, we can run a program with correct implementation, collect the states it produced and use them as reference to compare with the states that any student program produces. If they match, the student program is functionally correct.

The goal is to instrument the program to collect the states it goes through during execution. It is equally important to make the program complete without delays or stalls. This means that any user interaction (e.g., wait for user input) should be avoided. Ideally, one should get rid of user interface which is adding delay.

One of the big challenges is to guarantee that the program execution is always **`deterministic`**. It should not matter how many times we execute the program. Every execution must produce an identical result. To achieve the repeatable behavior, Delve's approach is to assist the user replace any source of randomness in the program.

This User Guide illustrates how to instrument the code using an example that falls in the hardest category. Yahtzee is a CS1 assignment with all the factors that make instrumentation difficult: **i) user interface** (slows down the execution), **ii) user input** (stalls the execution), **iii) randomness** (non-deterministic behavior). Note that code instrumentation is easier in assignments that lack one or more of those factors.

The remaining section covers the instrumentation process step-by-step.

Before the instrumentation the user has to set the timeout value, which is the amount of time (in seconds) to wait for the program complete and terminate it if still running. This is important to avoid situations of infinite-loops etc. The user can choose the proper value (or no timeout at all) based on the nature of the program.

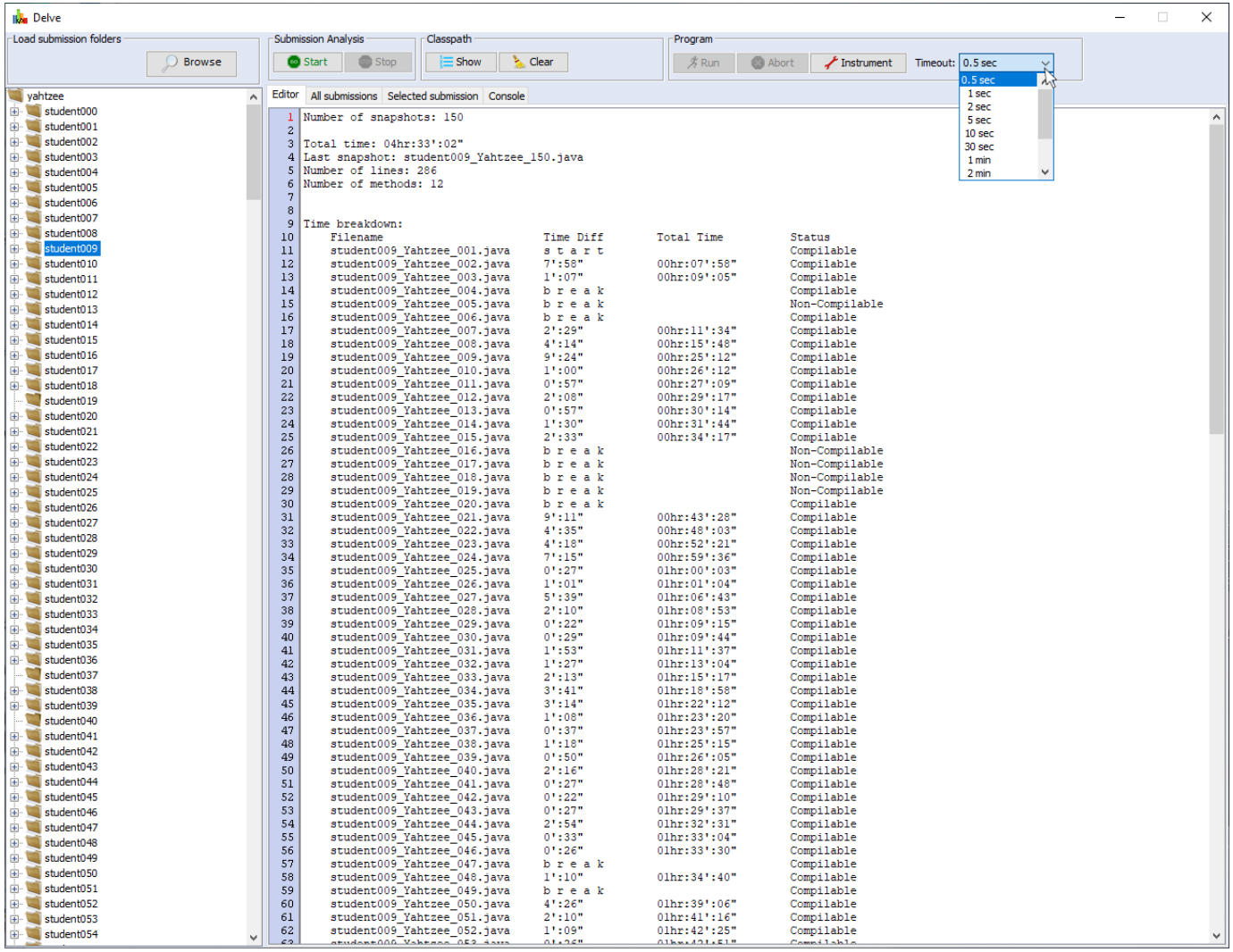

Figure 32:The user can select the amount of time to wait for the program to complete or else terminate it.

To initiate the instrumentation process, the user has to press button **`Instrument`**.

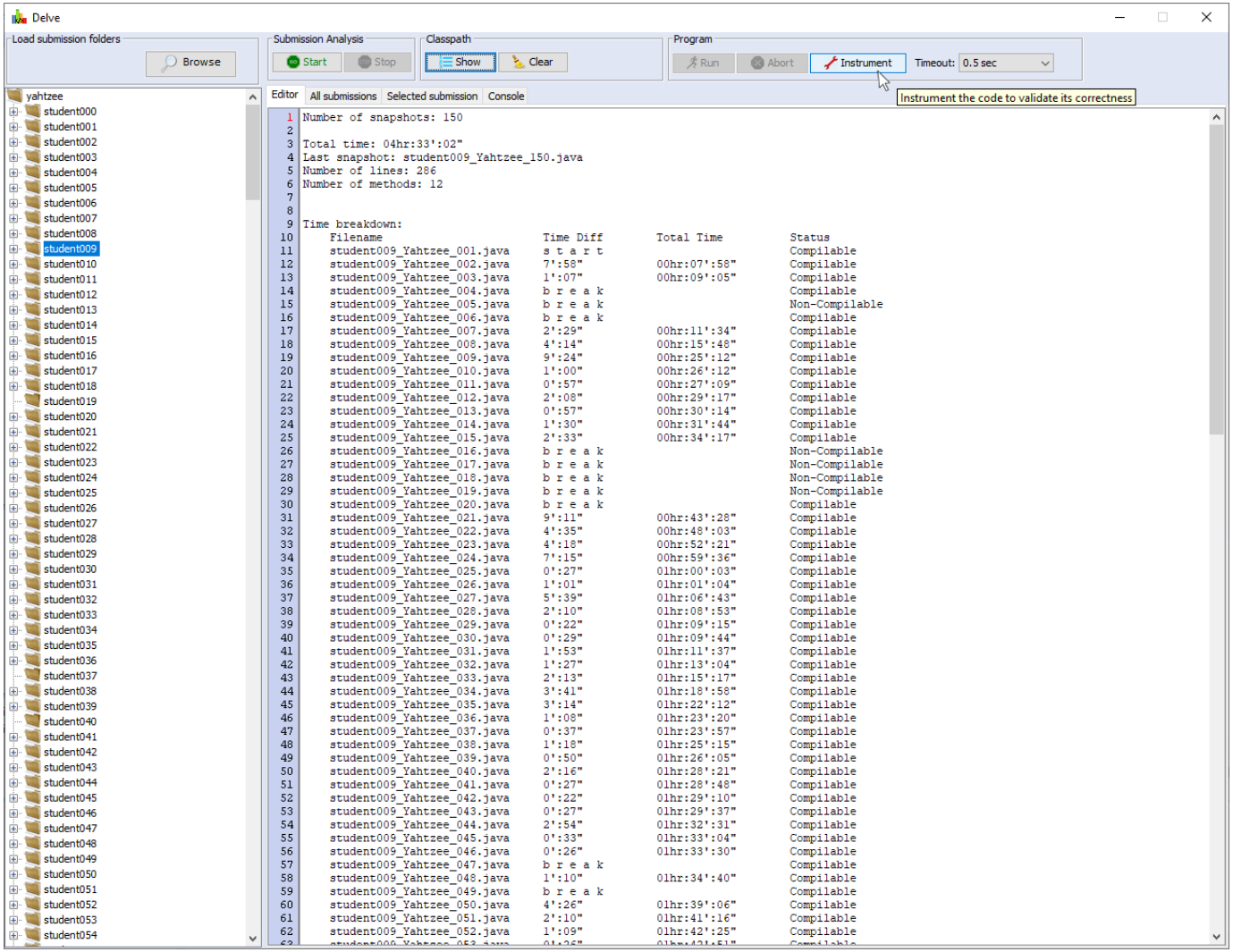

Figure 33: Pressing button Instrument initiates the instrumentation process.

7.1 Instrumentation Overview

At this point, a dialog window to guide the user shows up. The first screen lists the instrumentation steps.

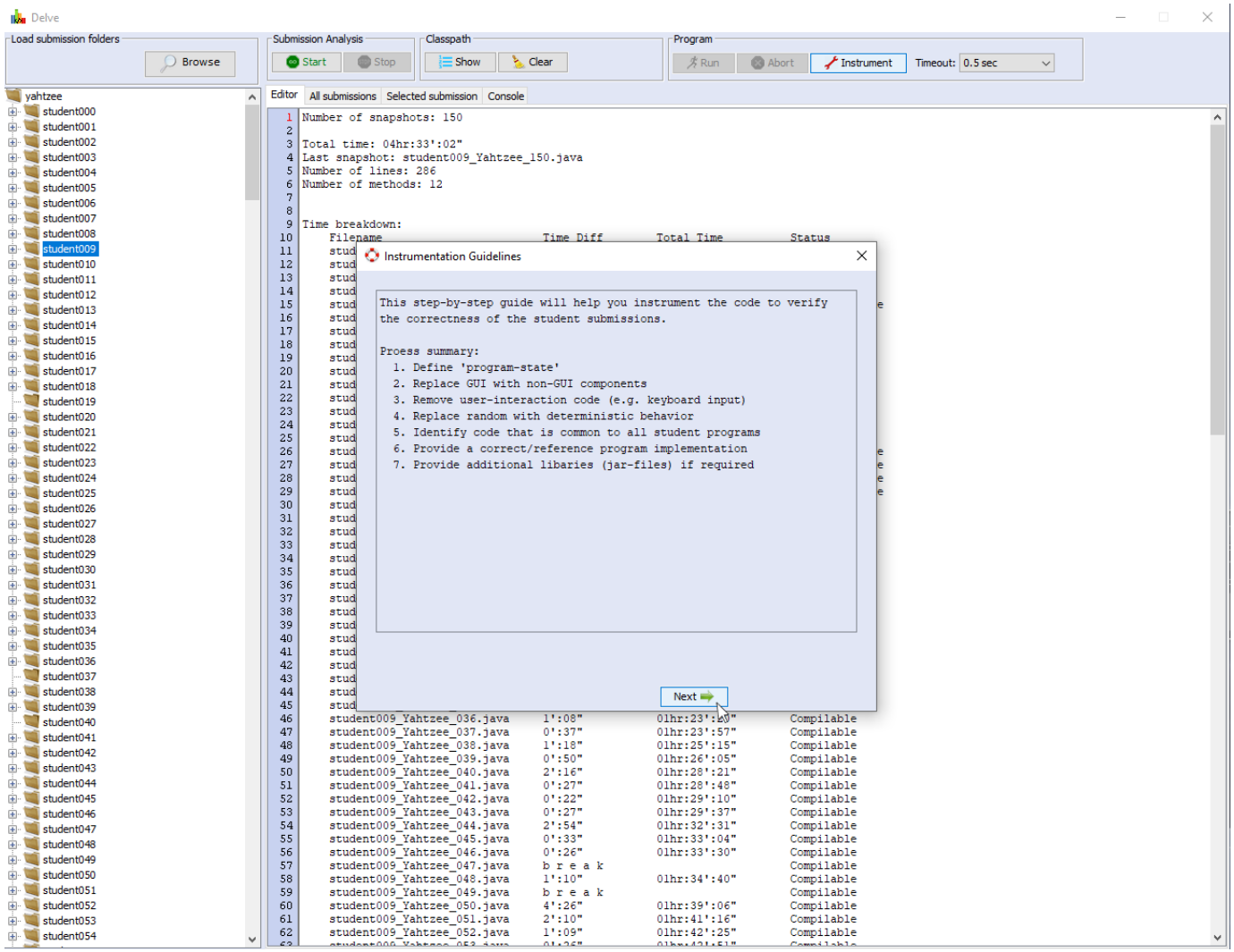

Figure 34: The first dialog screen with an overview of the instrumentation steps.

7.2 Step 1: Program State

The first step is to define what **`program state`** means for this assignment. This varies between different programs.

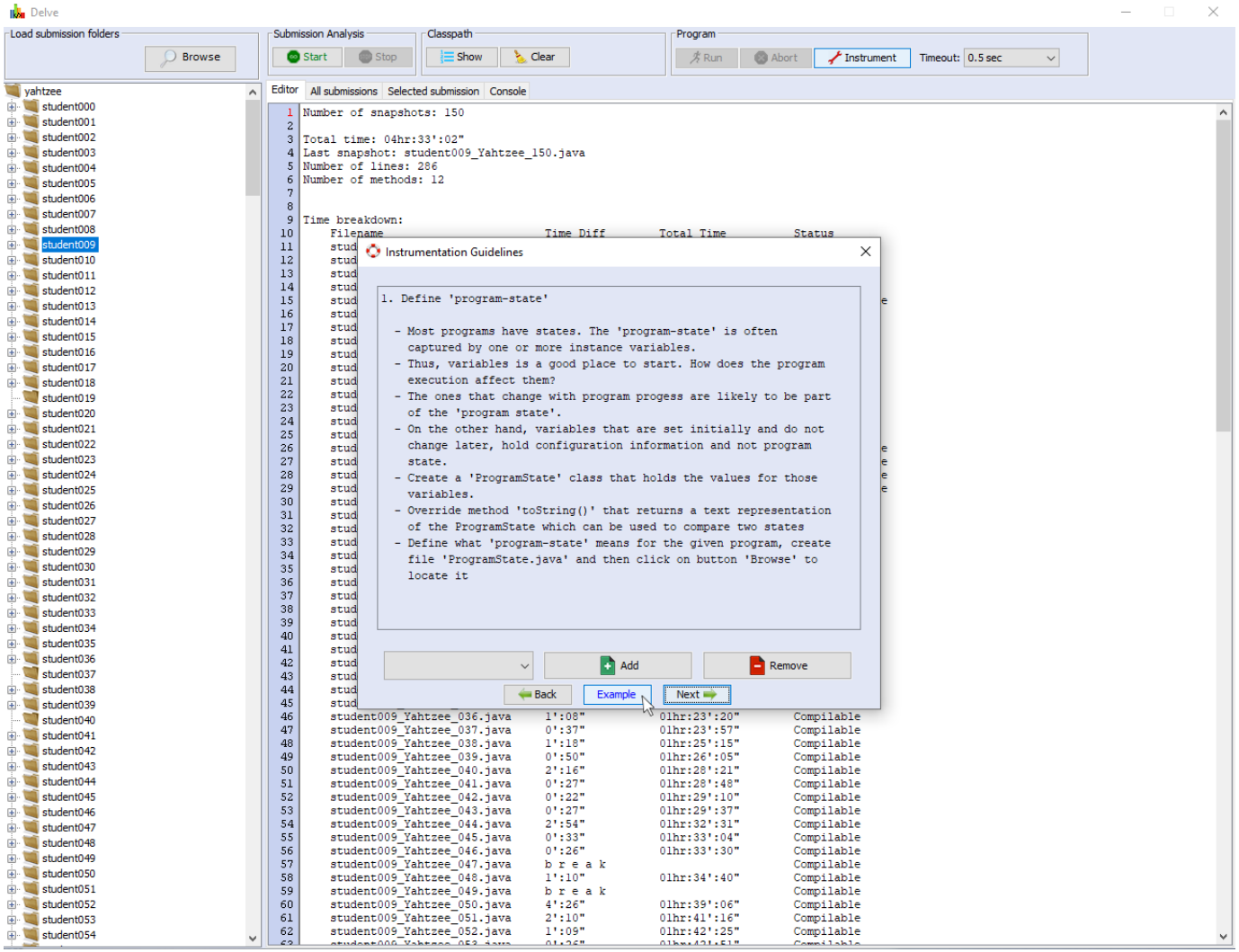

Figure 35: Delve provides a number of hints to help the user define the program state for the given assignment.

To get a better idea about the **`program state`**, the user can go over an example.

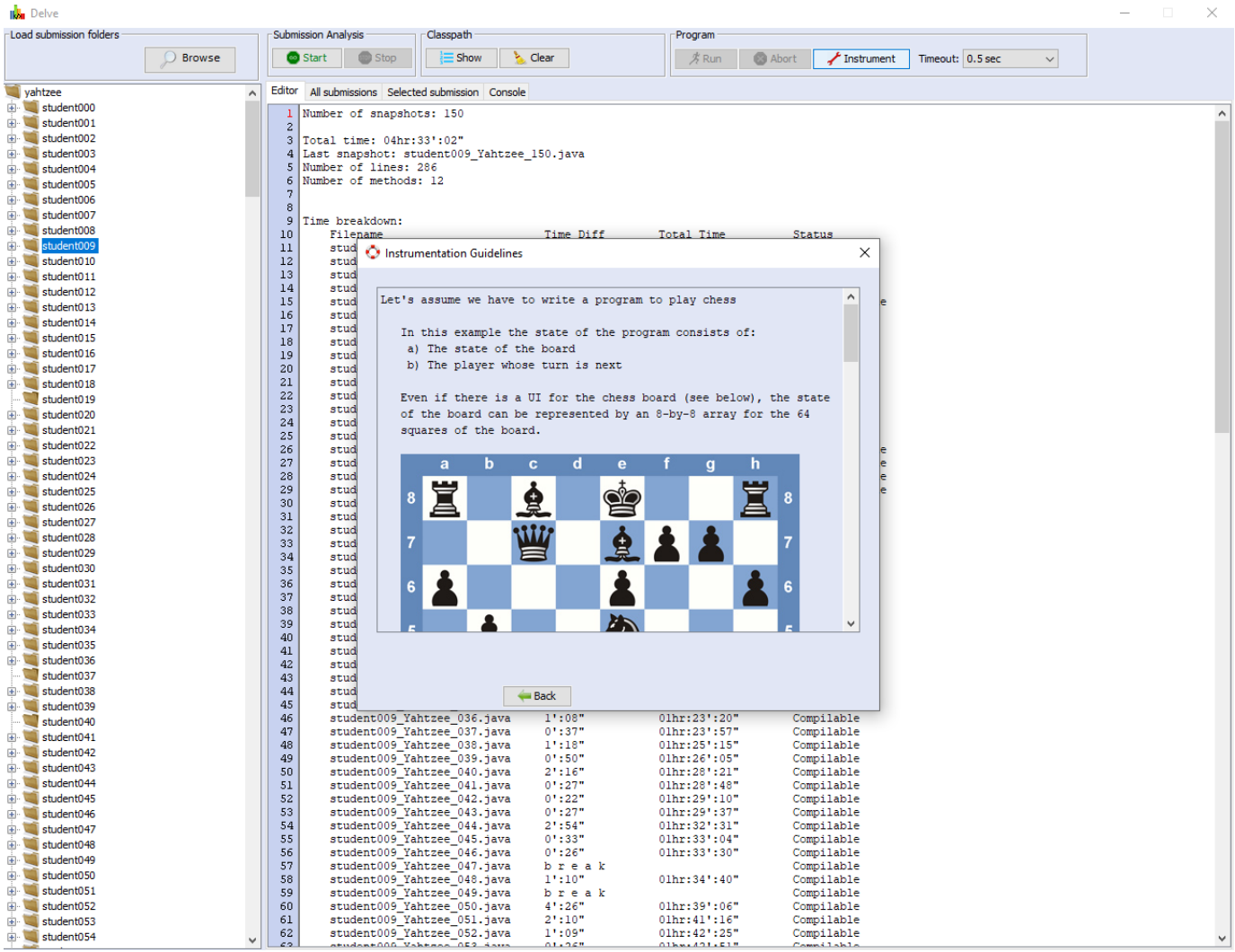

Figure 36: An example to explain what program state means including the source code.

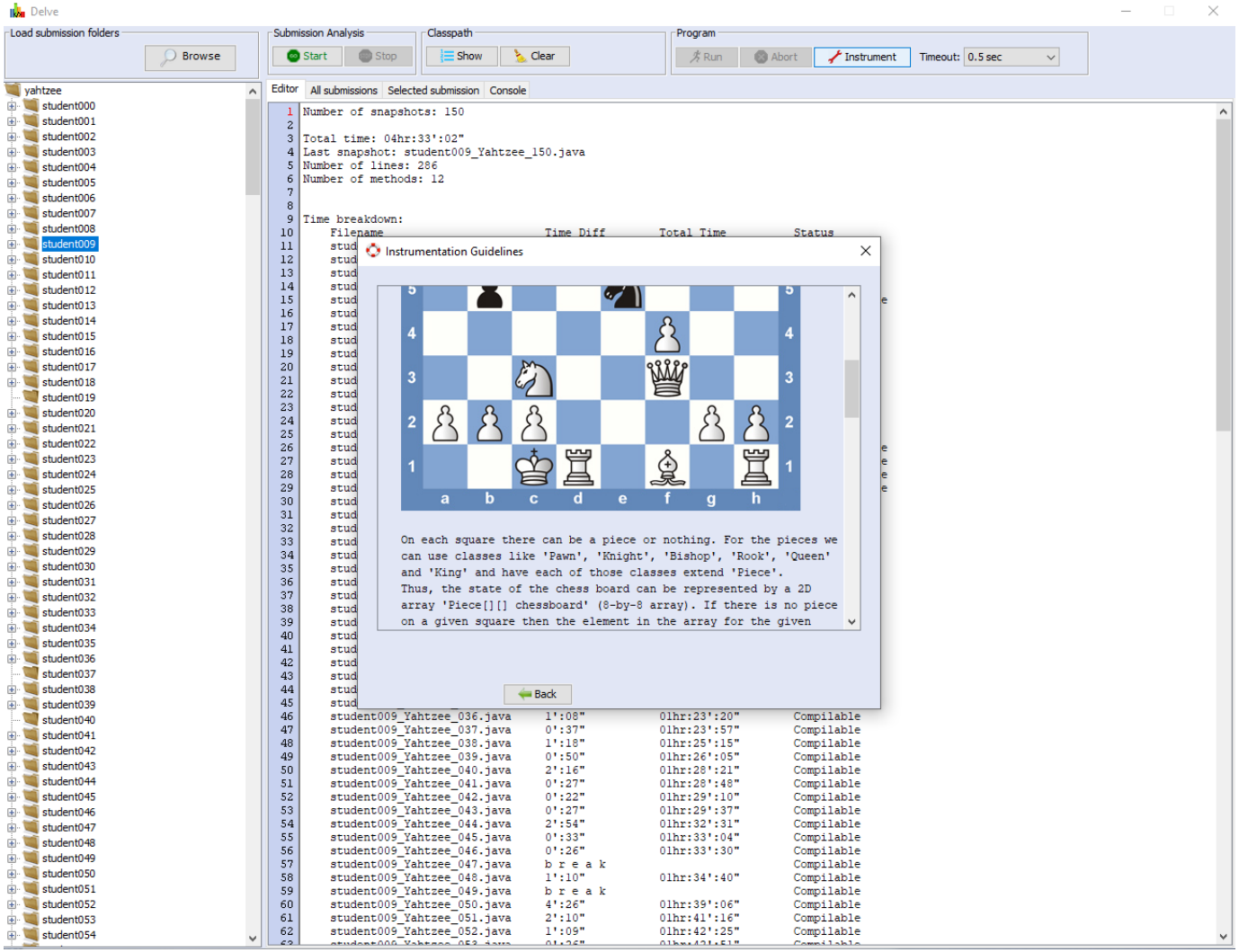

Figure 37: An example to explain what program state means including the source code.

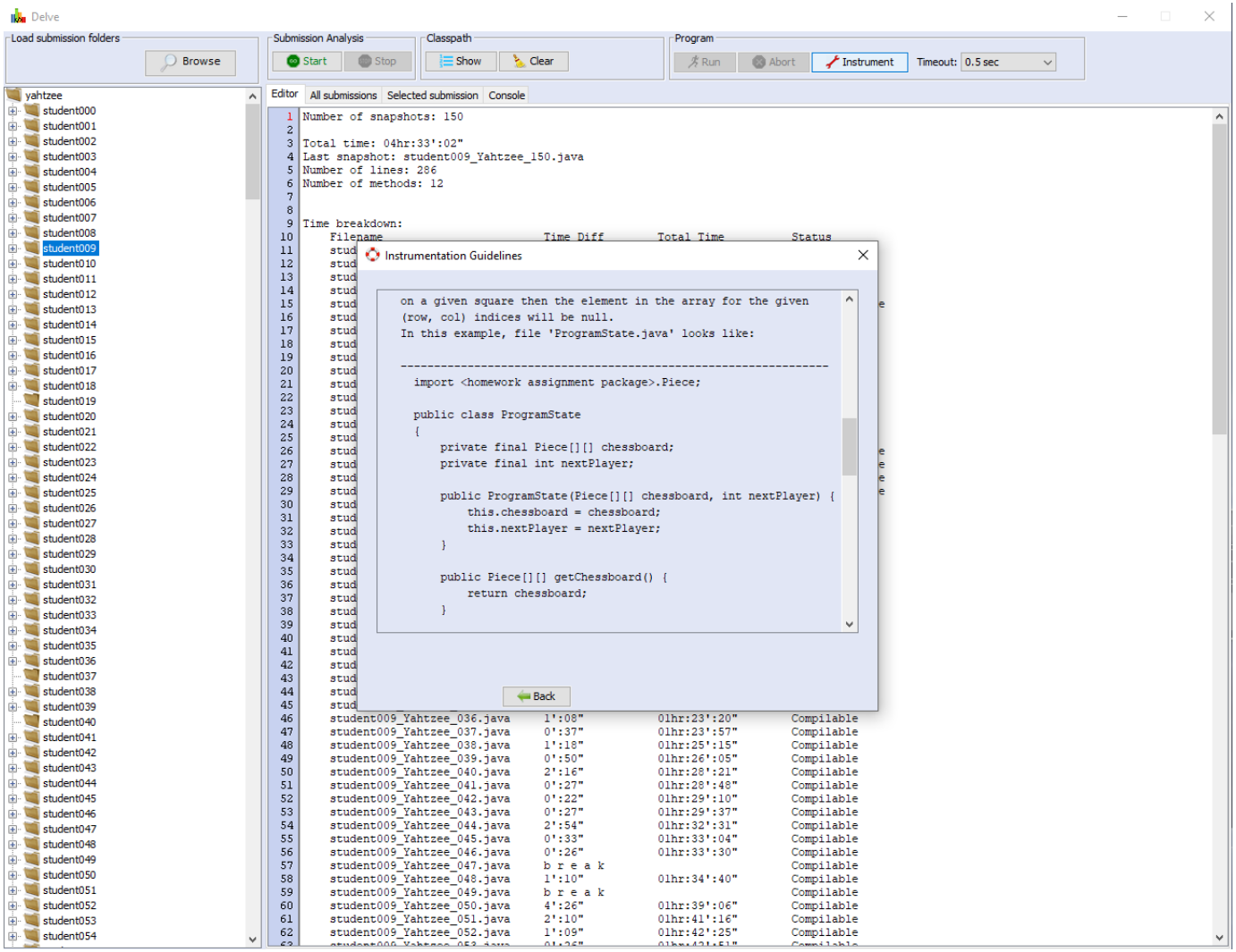

Figure 38: An example to explain what program state means including the source code.

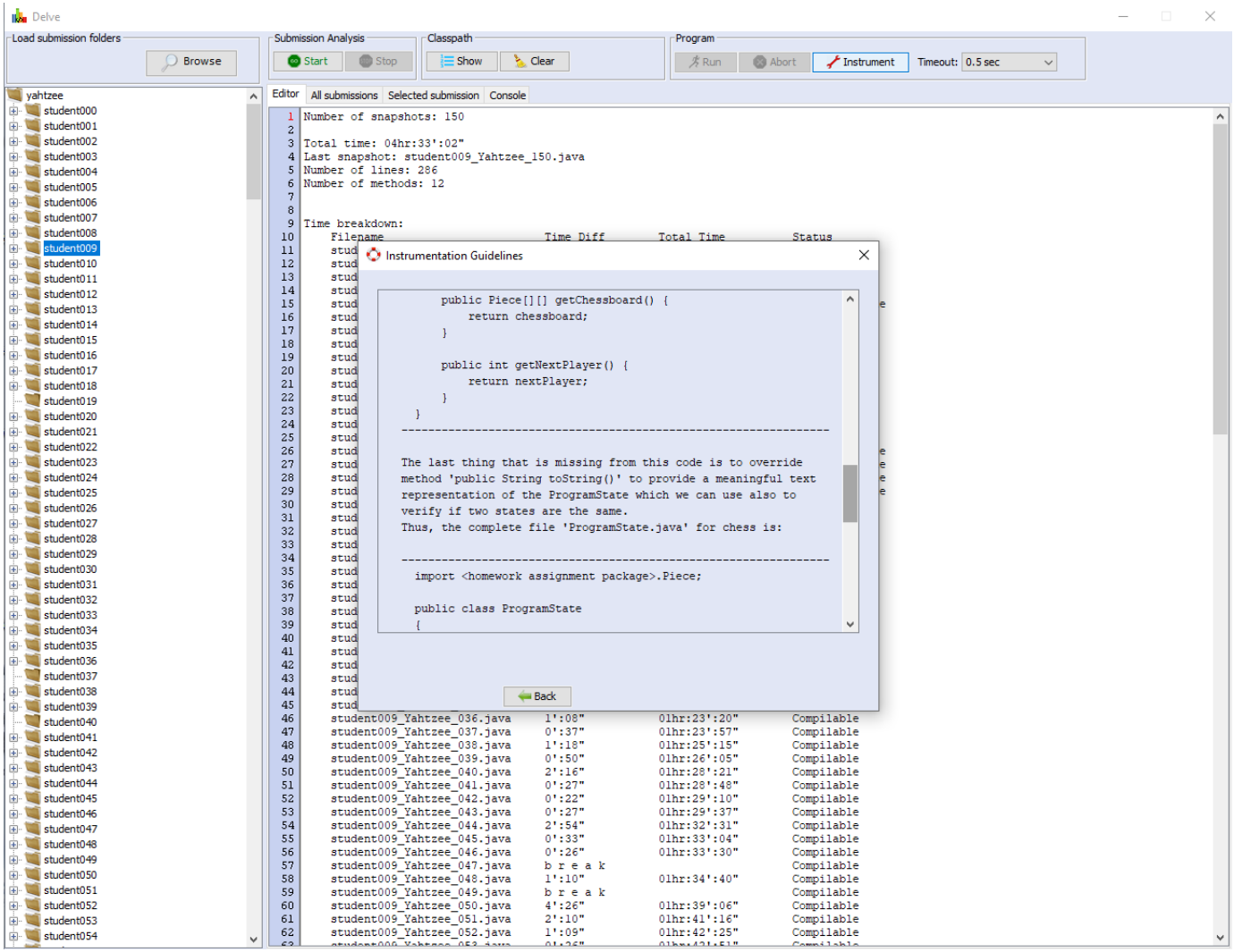

Figure 39: An example to explain what program state means including the source code.

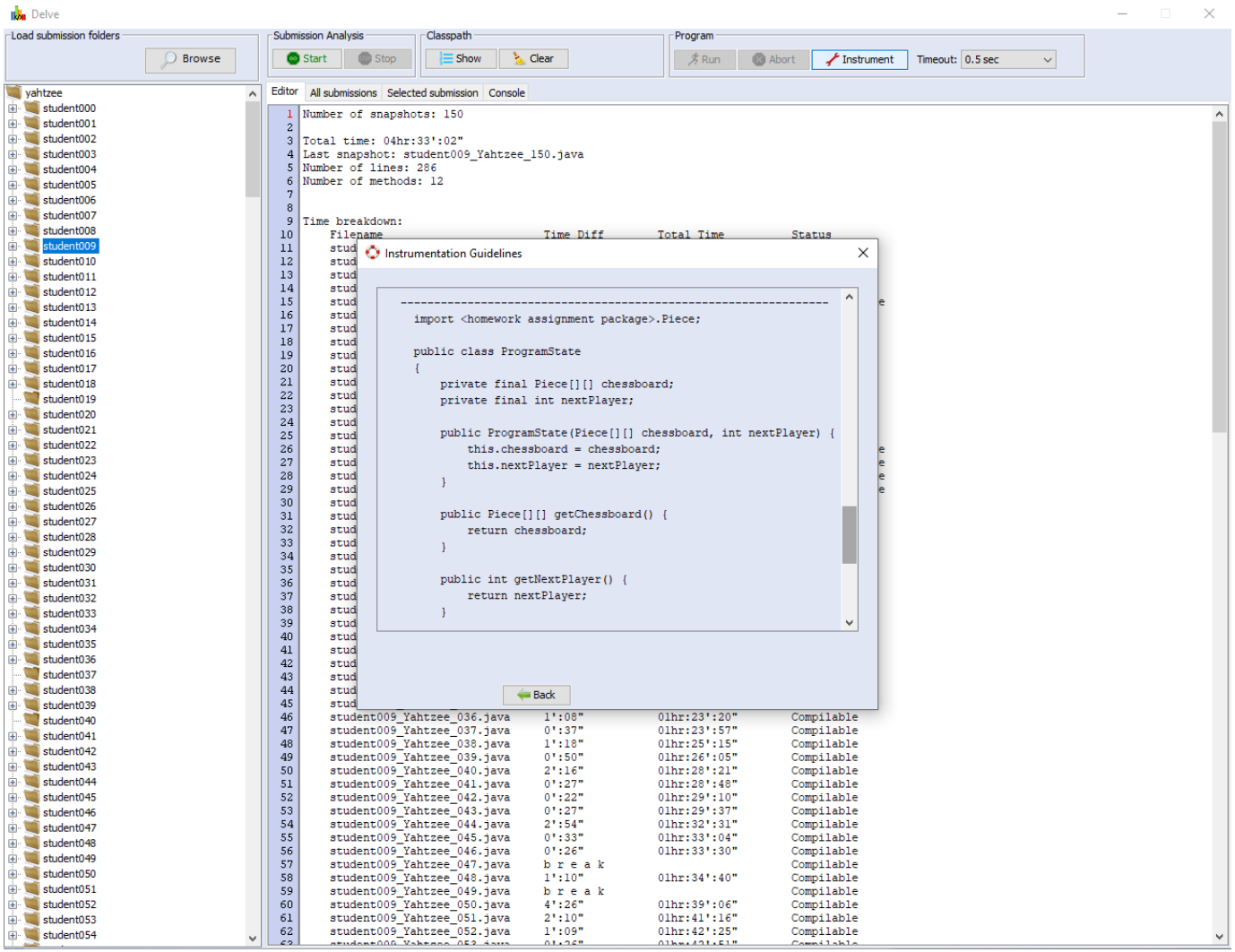

Figure 40: An example to explain what program state means including the source code.

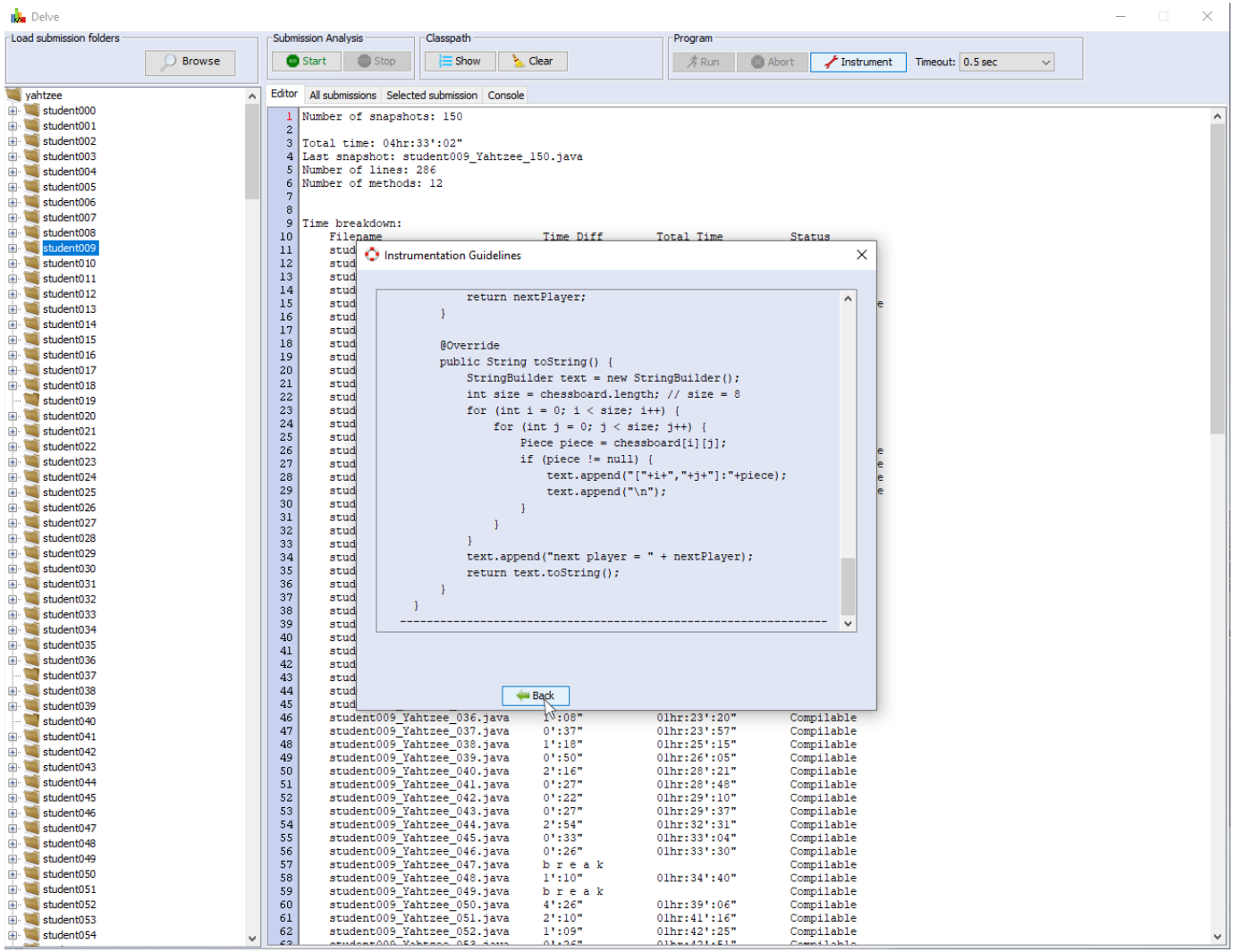

Figure 41: An example to explain what program state means including the source code.

To add the code definition for the **`program state`**, the user can click the **`Add`** button

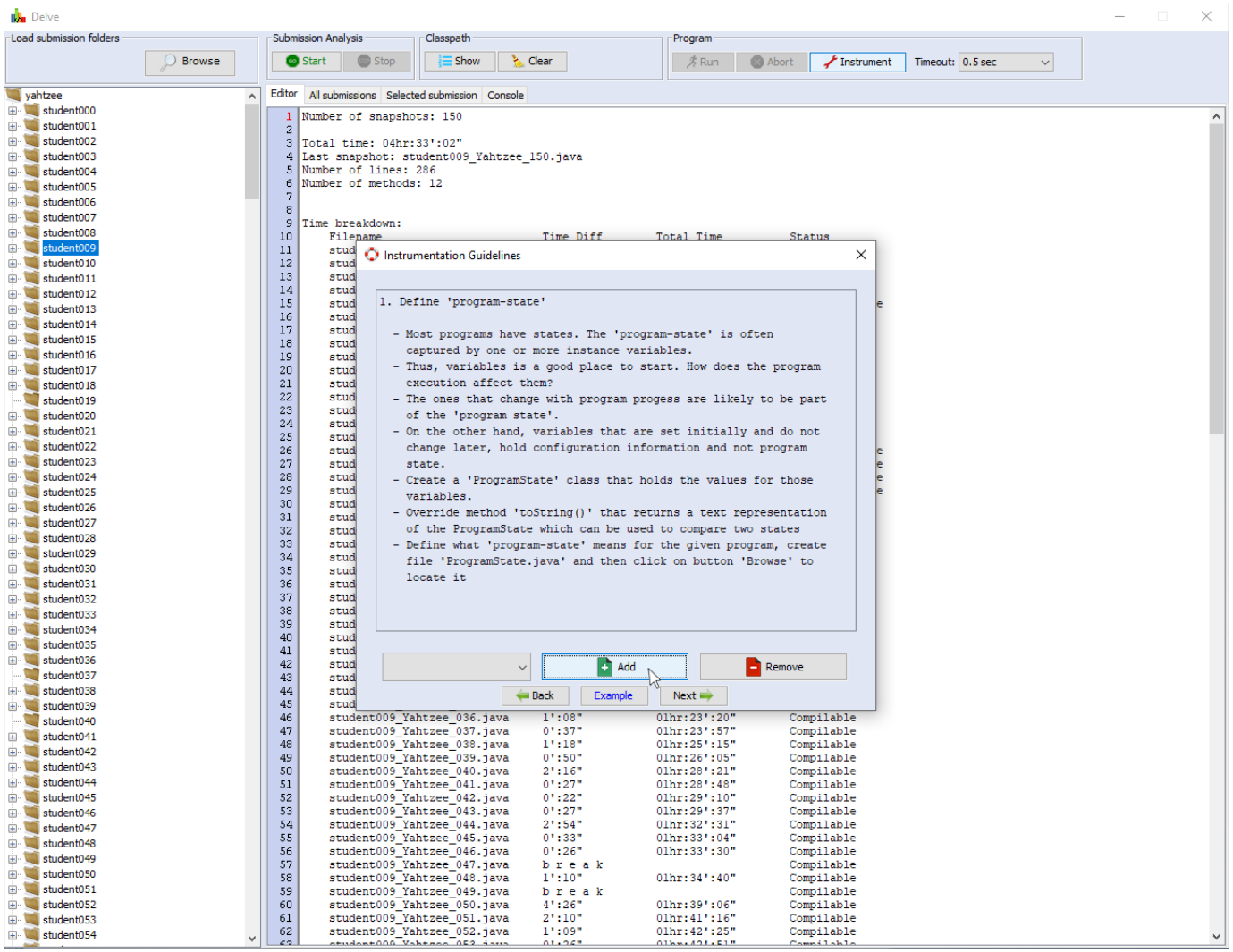

Figure 42: Click the *Add* button to locate the file with the code definition for the *program state*.

and then locate that file (i.e., **`ProgramState.java`**).

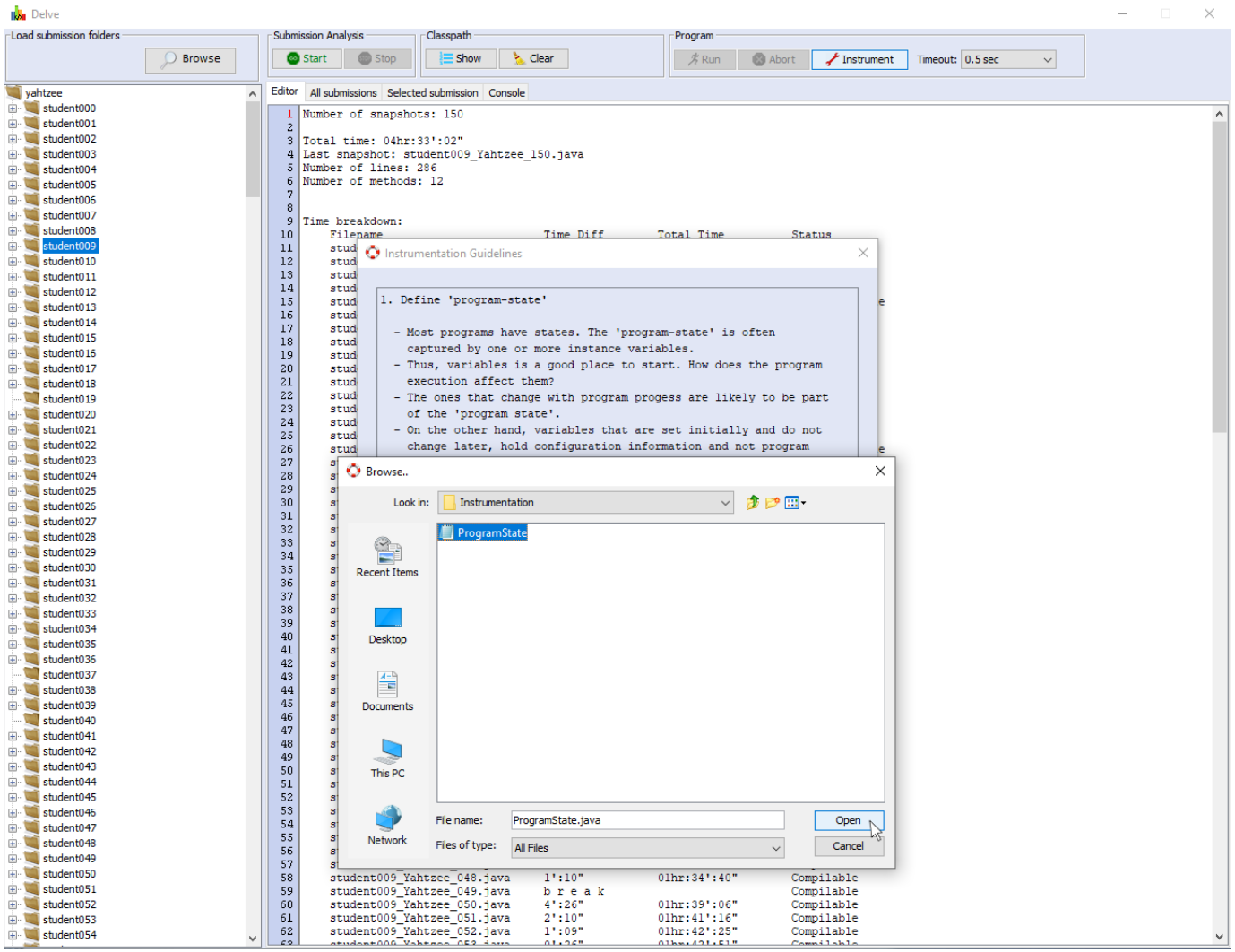

Figure 43: The user can now locate the file with the code for the program state.

This code can contain any package declaration (in this example *'package mycode;'*). The only required method is **`public String toString()`** which is used to compare a state with another.

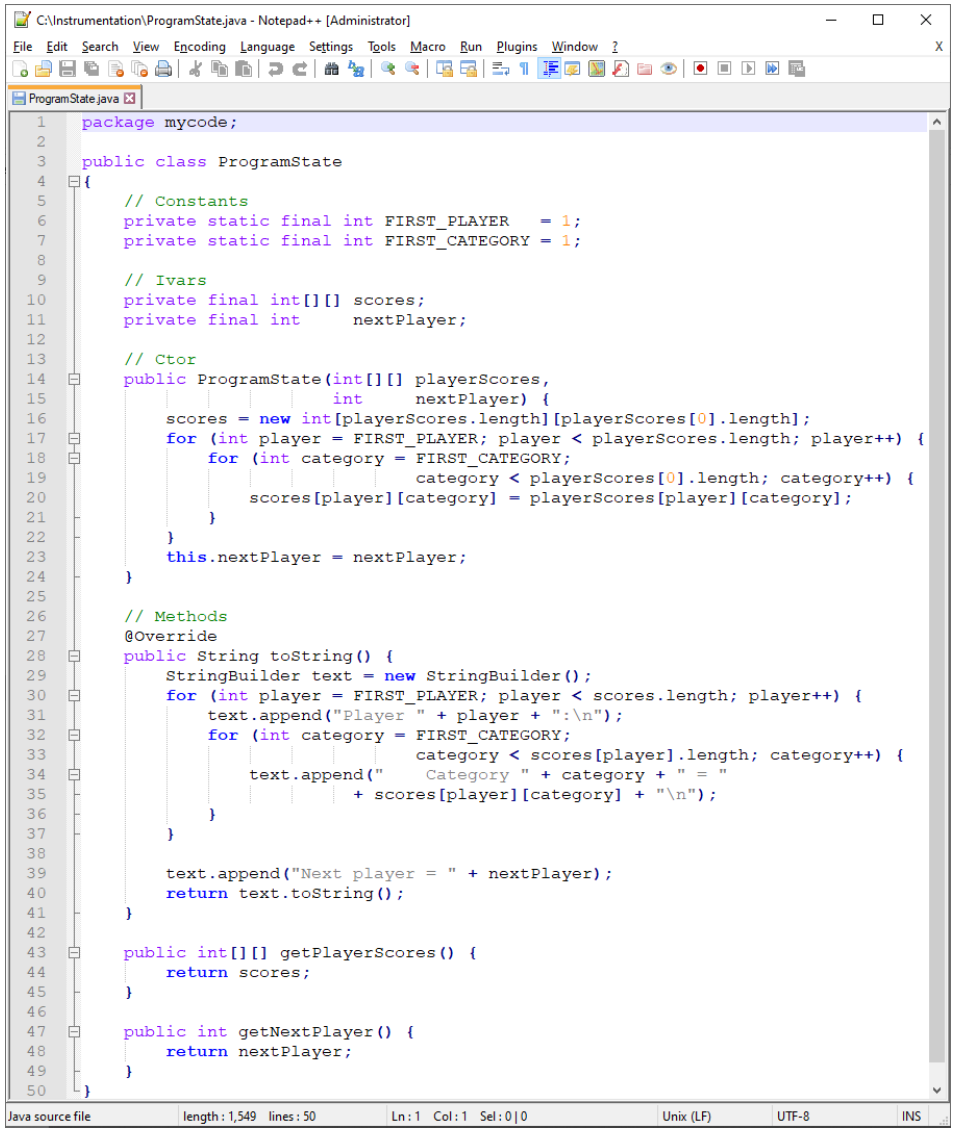

Figure 44: The code for the *program state* in our example.

At this point we have completed the first step. Note that this step is required for instrumentation.

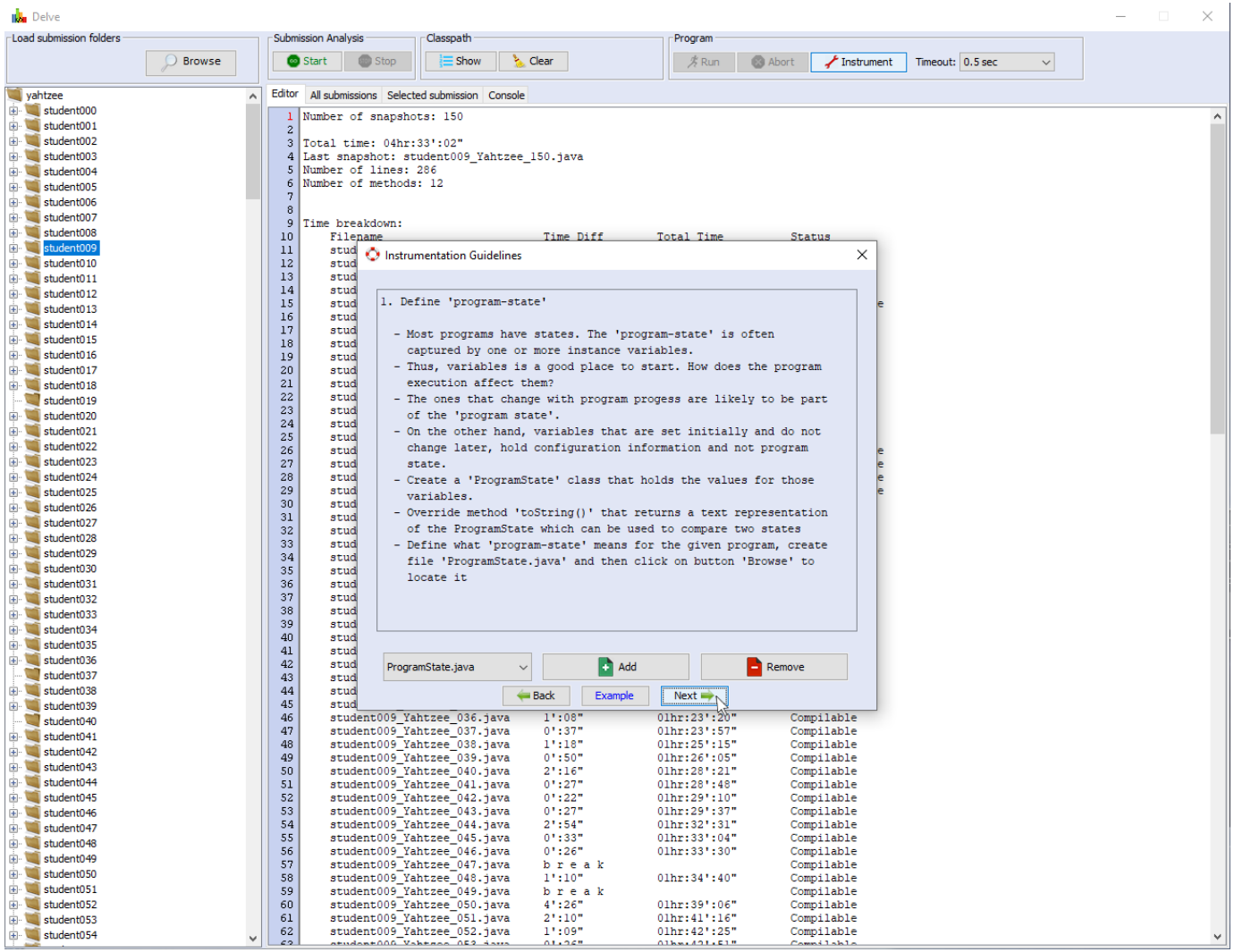

Figure 45: The first step is complete. Ready to move on to the second step.

7.3 Step 2: Replace GUI with non-GUI components (bypass UI behavior)

The second step is optional and in fact not even required for programs without a UI. However, it is highly recommended.

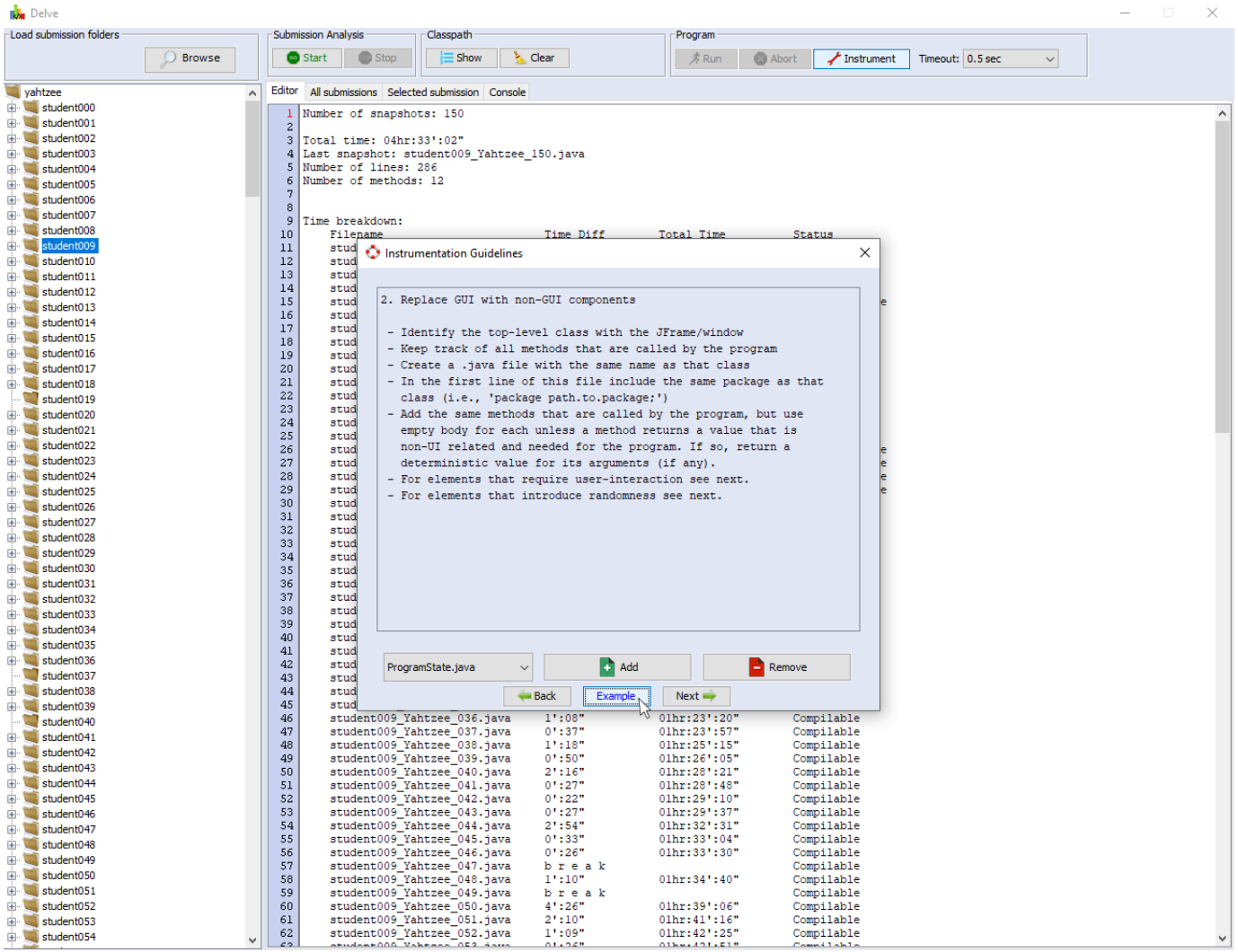

Figure 46: Second step: Replacing GUI with non-GUI components.

Again, Delve offers an example to illustrate how this works in practice.

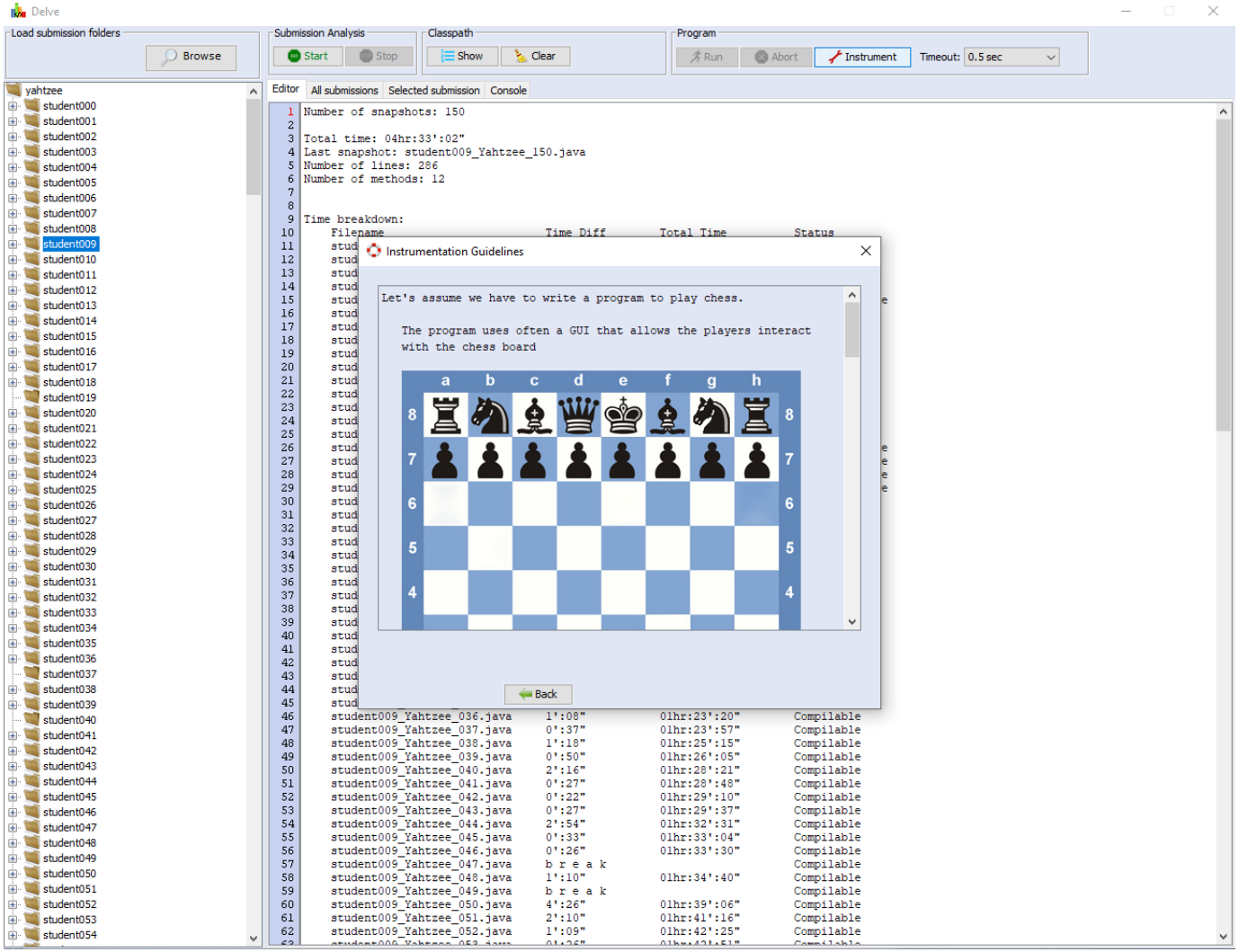

Figure 47: An example that illustrates how to replace GUI with non-GUI components.

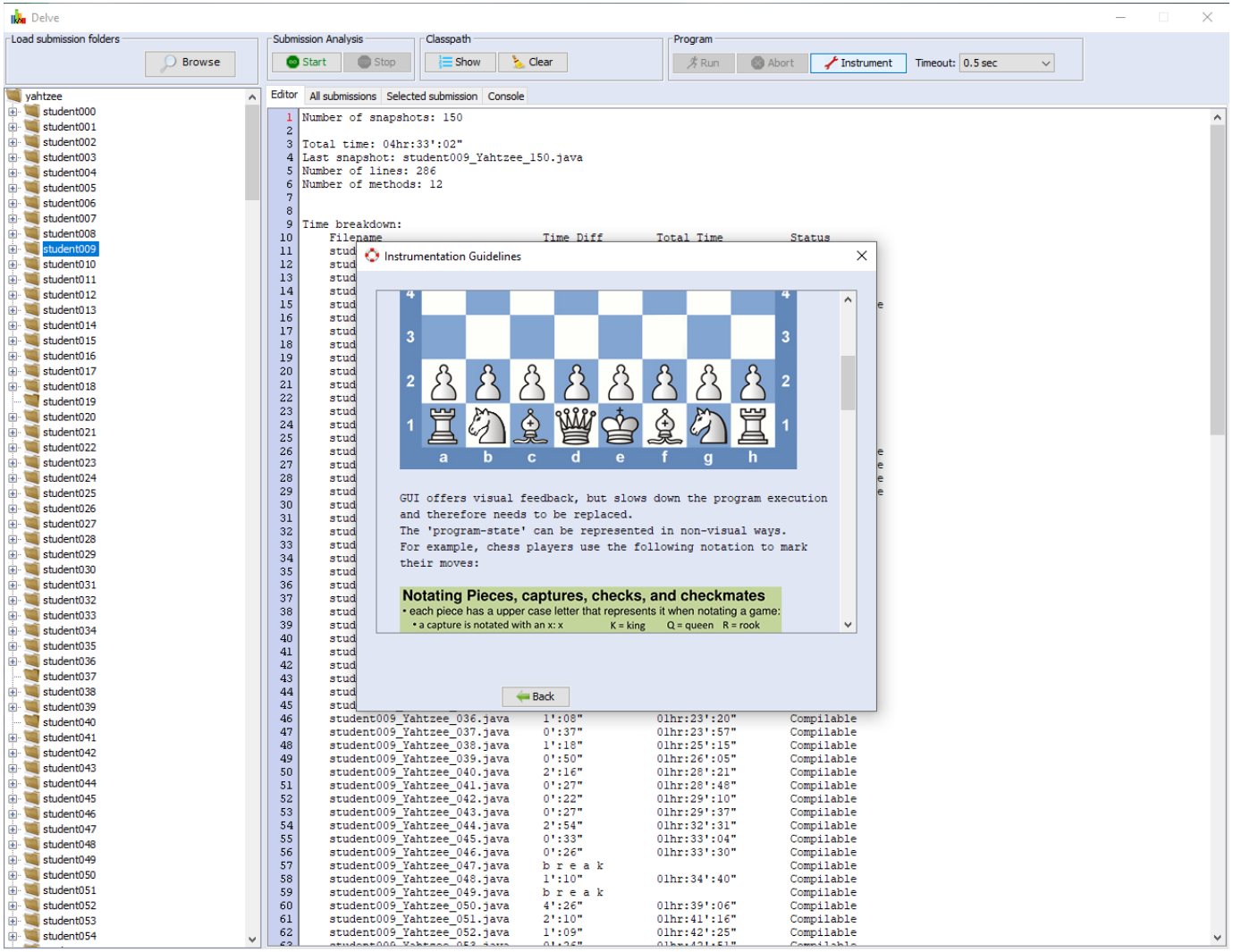

Figure 48: An example that illustrates how to replace GUI with non-GUI components.

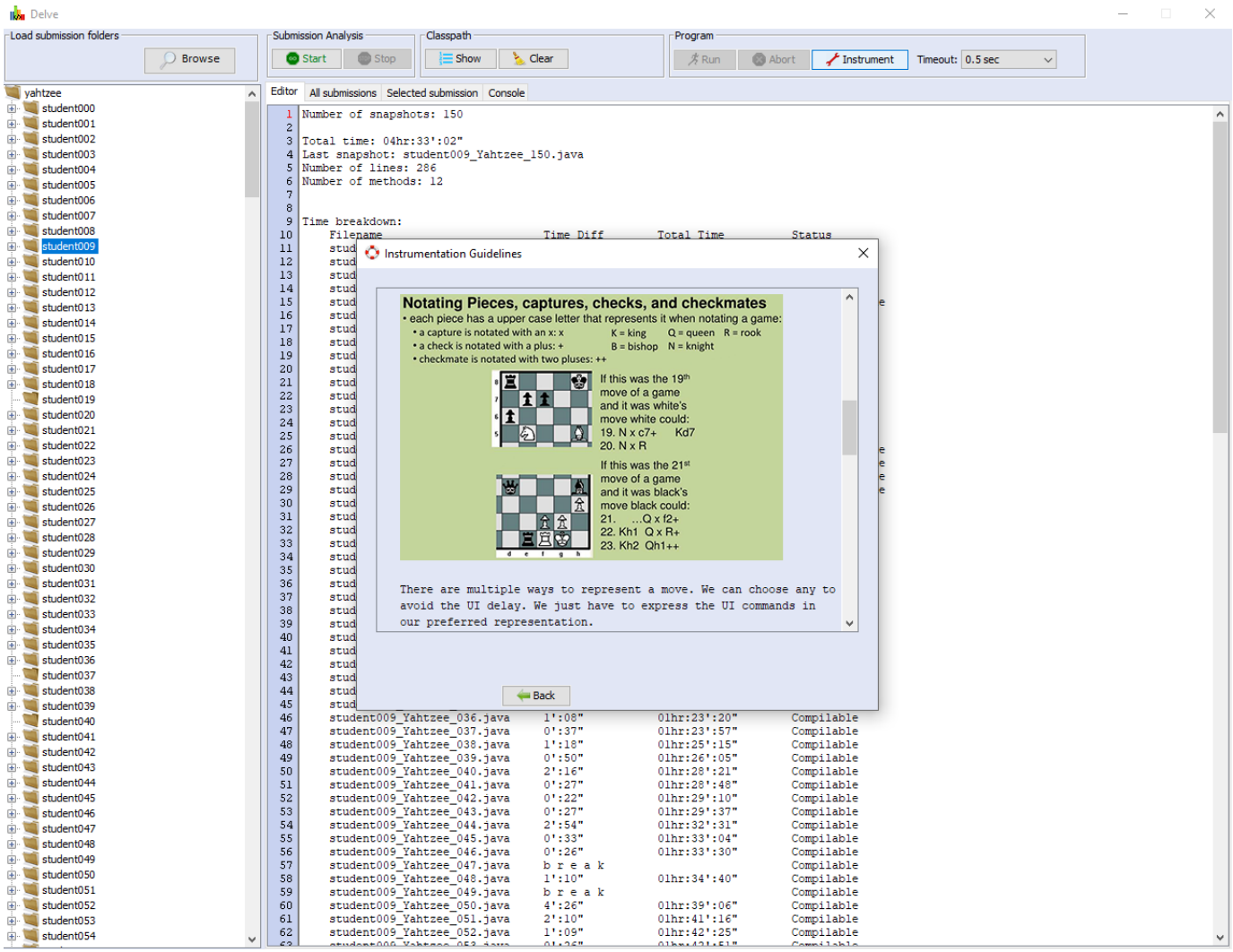

Figure 49: An example that illustrates how to replace GUI with non-GUI components.

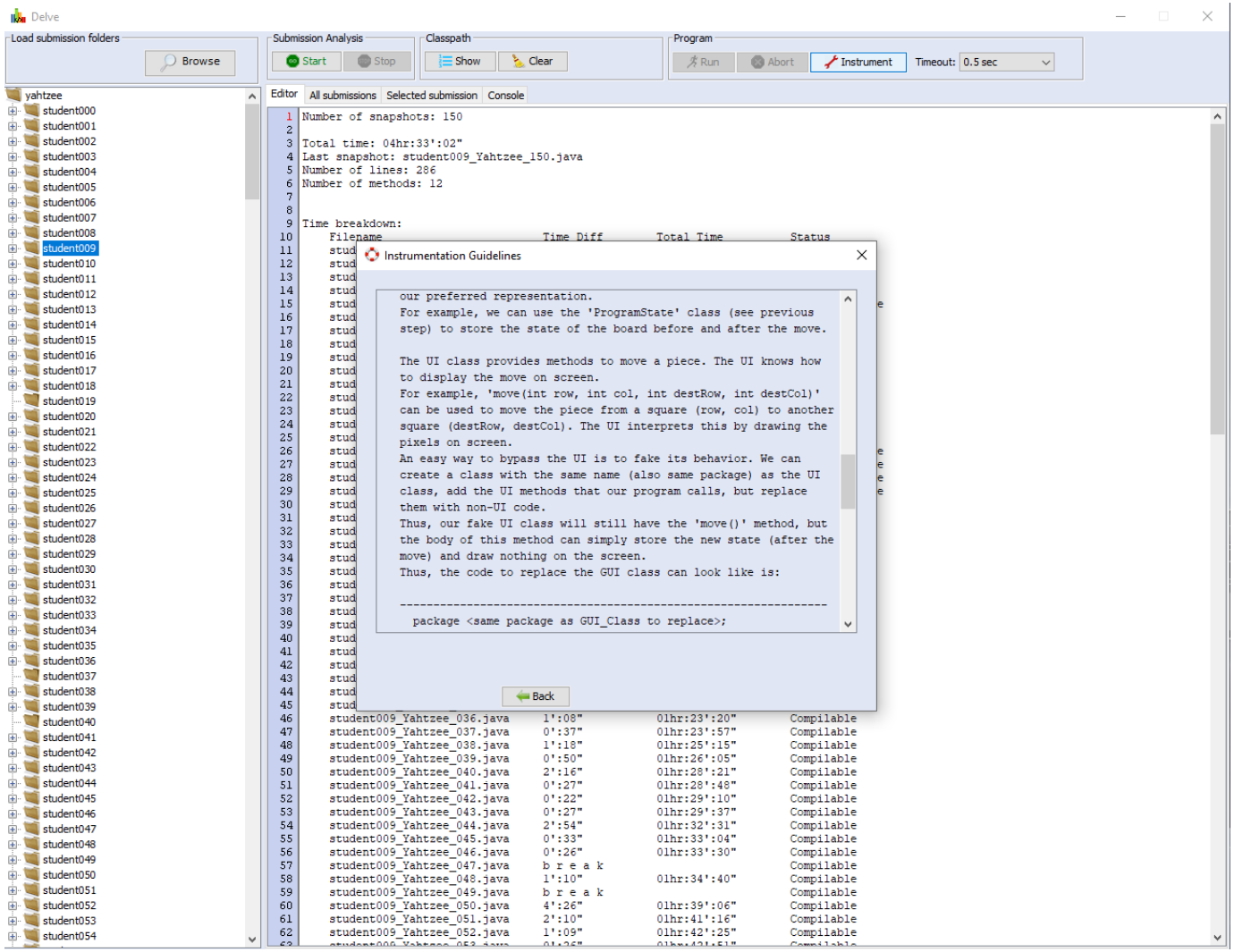

Figure 50: An example that illustrates how to replace GUI with non-GUI components.

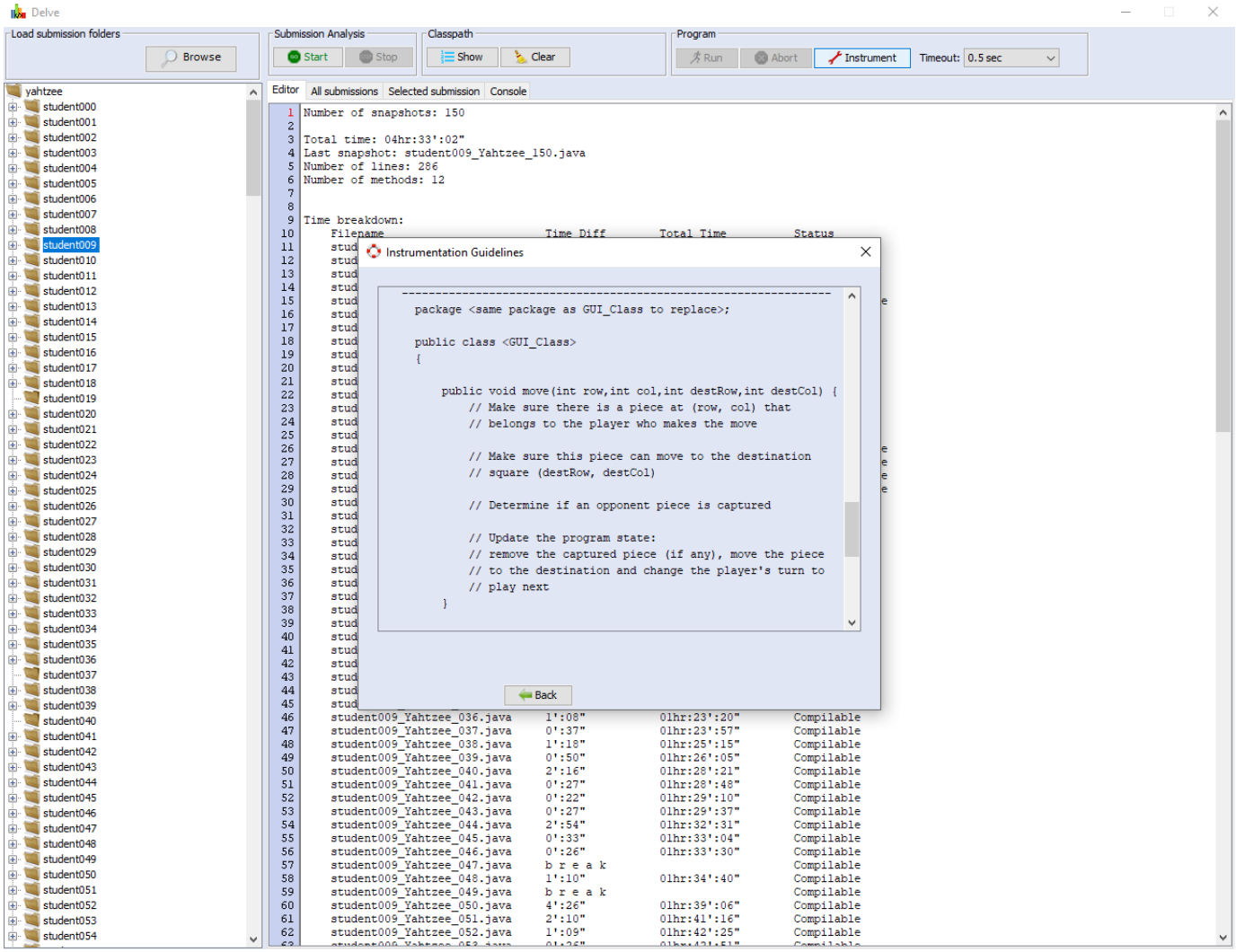

Figure 51: An example that illustrates how to replace GUI with non-GUI components.

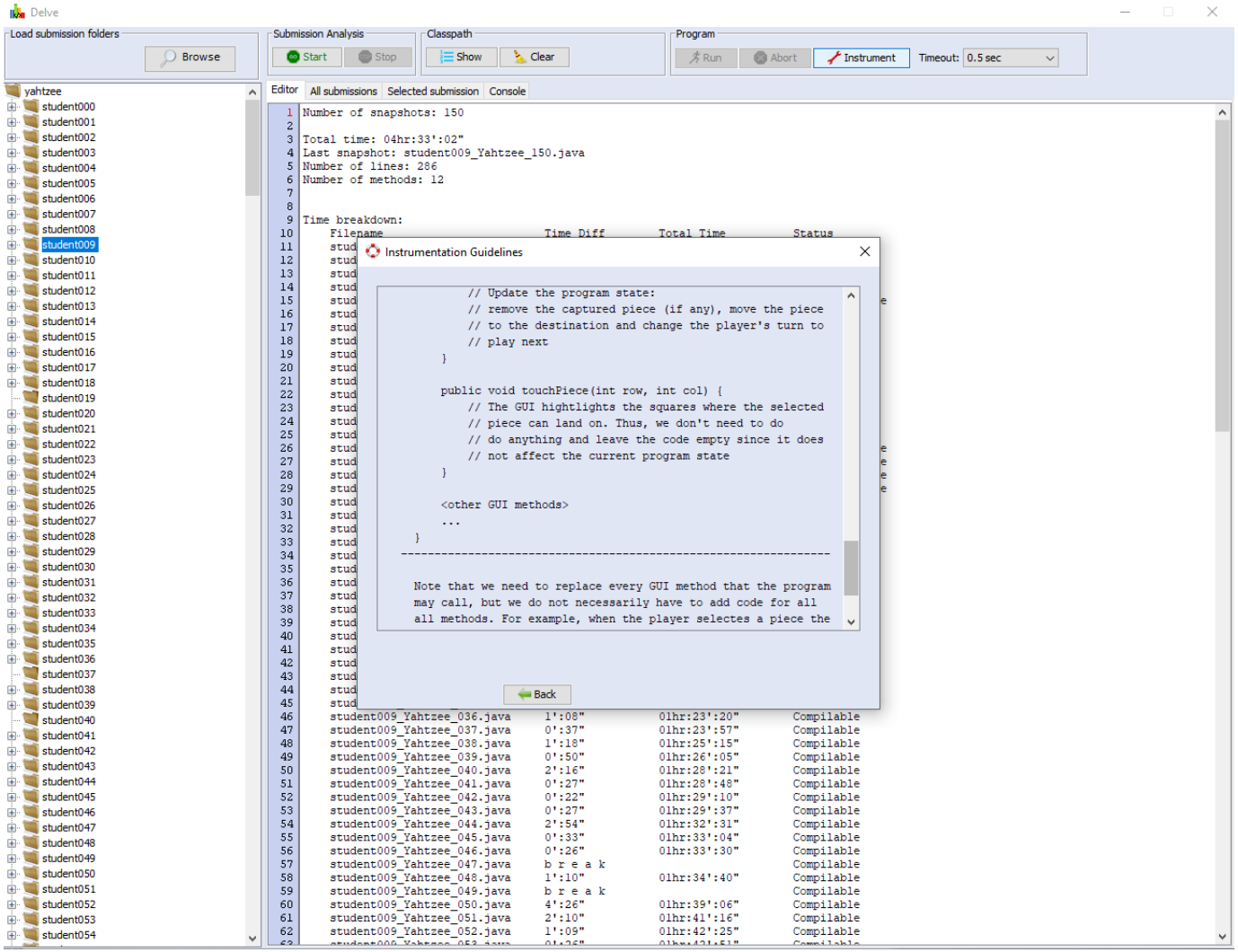

Figure 52: An example that illustrates how to replace GUI with non-GUI components.

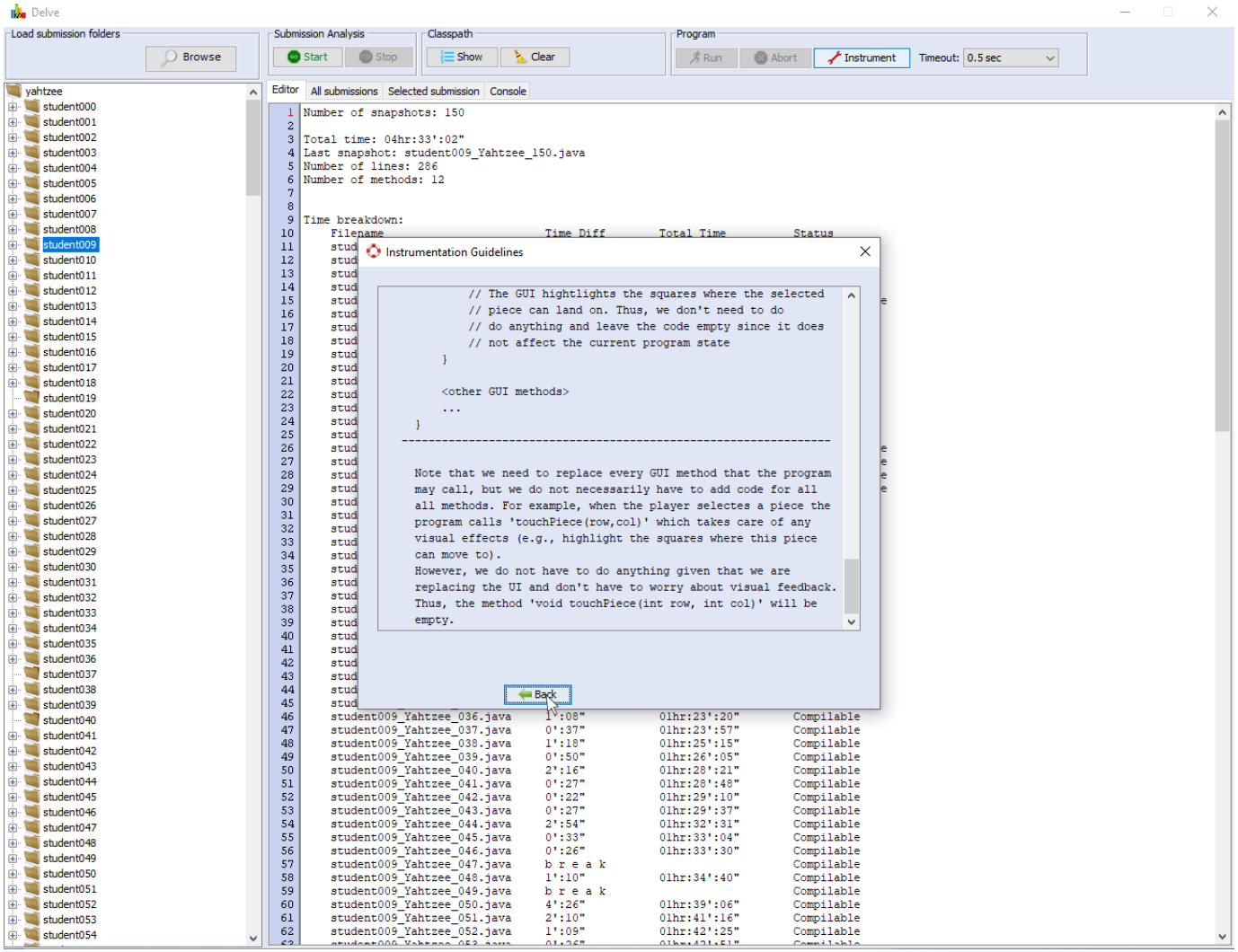

Figure 53: An example that illustrates how to replace GUI with non-GUI components.

To add the code to avoid UI delays, the user can click the **Add** button

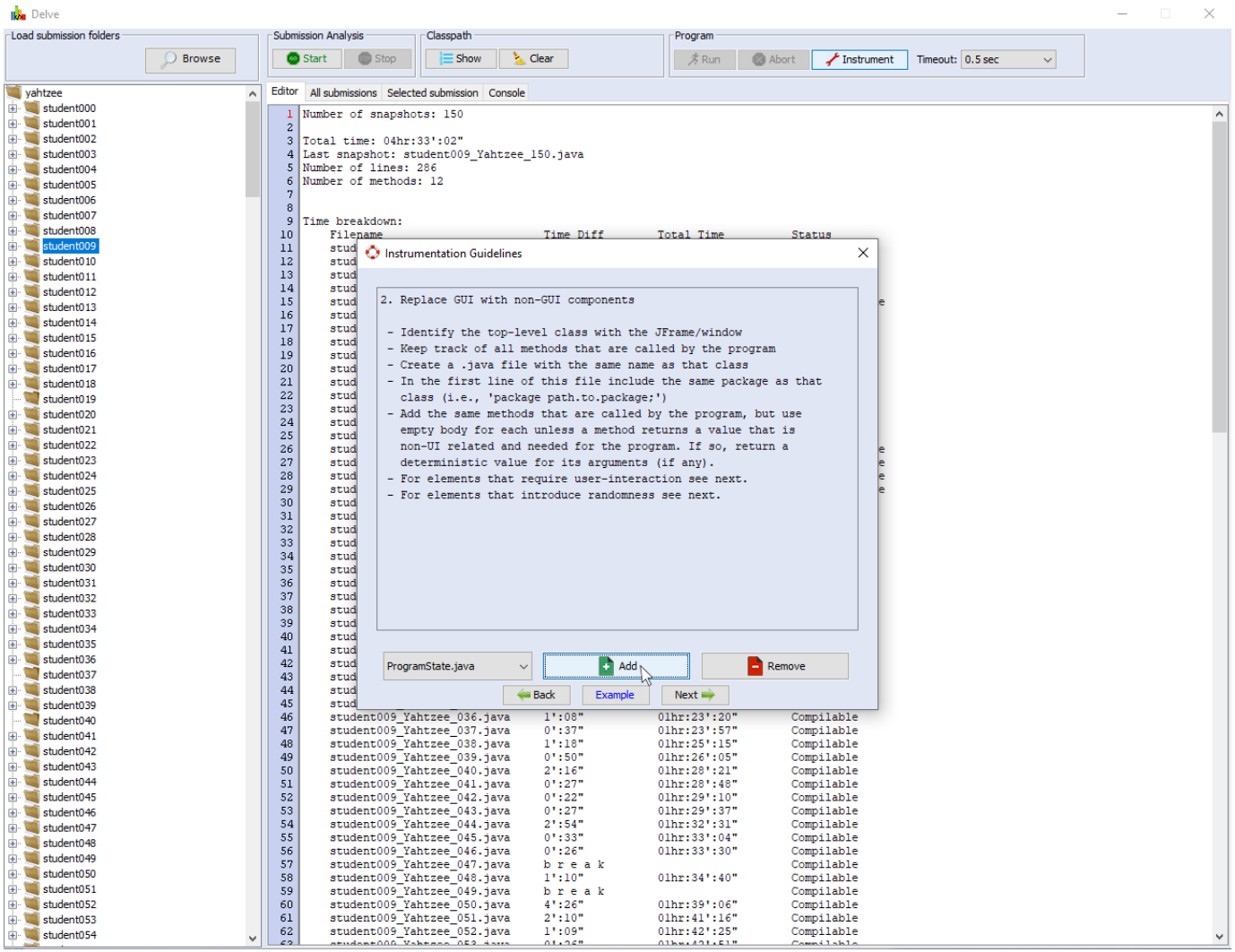

Figure 54: Click the *Add* button to locate the file with the code to avoid UI delays.

and then locate the file(s) to replace the GUI components.

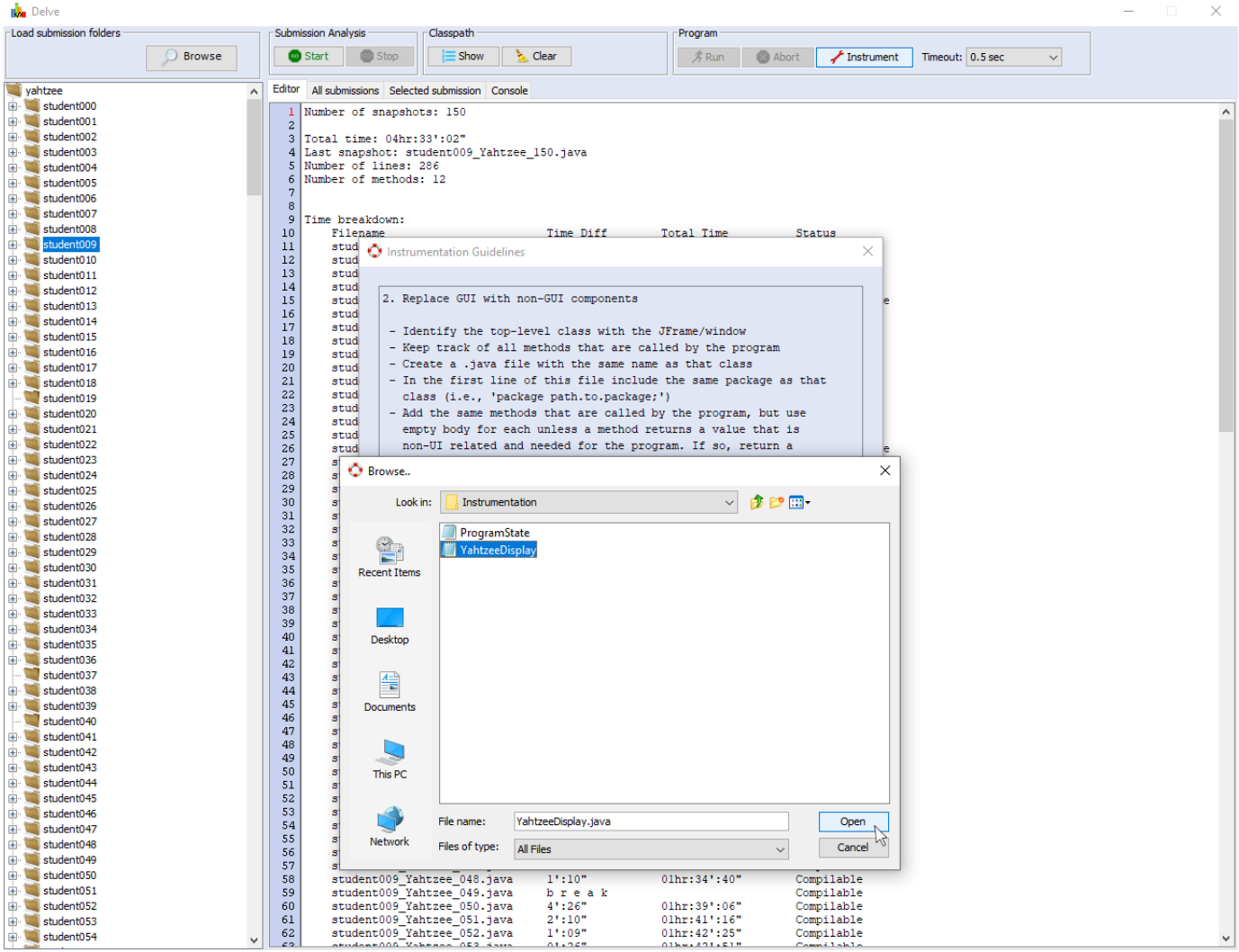

Figure 55: The user can now locate the file with the module that replaces the first UI component.

Most of the UI replacement code is actually doing nothing. We can take advantage of the fact that every program is making UI calls and use some methods to save the program state (we will revisit this in Section **7.6**).

C:\Instrumentation\YahtzeeDisplay.java - Notepad++ [Administrator]

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

YahtzeeDisplay.java

```
1  import java.awt.Container;
2  import mycode.ProgramState;
3
4  public class YahtzeeDisplay implements YahtzeeConstants
5  {
6      // Constants
7      private static final int NUM_OF_PLAYERS = 3;
8      private static final int FIRST_PLAYER   = 1;
9      private static final int FIRST_CATEGORY = 1;
10     public static final int[] ROUNDS  = new int[] {
11       ONES, TWOS, THREES, FOURS, FIVES, SIXES, THREE_OF_A_KIND, FOUR_OF_A_KIND,
12       FULL_HOUSE, SMALL_STRAIGHT, LARGE_STRAIGHT, YAHTZEE, CHANCE };
13
14     // Ivars
15     private final int[][] scores;
16     private final int[] currRound;
17     private int currPlayer;
18
19     // Ctors
20     public YahtzeeDisplay(Container container, String[] playerNames) {
21         this(playerNames);
22     }
23     public YahtzeeDisplay(String[] playerNames) {
24         currRound = new int[NUM_OF_PLAYERS + FIRST_PLAYER];
25         scores = new int[NUM_OF_PLAYERS + FIRST_PLAYER][N_CATEGORIES + FIRST_CATEGORY];
26         for (int player = FIRST_PLAYER;
27                  player < NUM_OF_PLAYERS + FIRST_PLAYER; player++) {
28             currRound[player] = -1; // Not started the game yet
29             for (int category = FIRST_CATEGORY;
30                      category < N_CATEGORIES + FIRST_CATEGORY; category++) {
31                 scores[player][category] = 0;
32             }
33         }
34     }
35
36     // Methods
37     public void updateScorecard(int category, int player, int score) {
38         scores[player][category] = score;
39         int nextPlayer = (player < NUM_OF_PLAYERS)? player + 1 : 1;
40         ProgramState programState = new ProgramState(scores, nextPlayer);
41         InstrumentedProgram.setProgramState(programState);
42     }
43     public void waitForPlayerToClickRoll(int player) {
44         currPlayer = player;
45         currRound[currPlayer]++;
46     }
47     public int waitForPlayerToSelectCategory() {
48         return ROUNDS[currRound[currPlayer]];
49     }
50     public boolean isDieSelected(int index) { return false; }
51     public void printMessage(String message) {}
52     public void displayDice(int[] dice) {}
53     public void waitForPlayerToSelectDice() {}
54 }
```

Java sou length : 2,106 lines : 54 | Ln : 1 Col : 1 Sel : 0 | 0 | Windows (CR LF) | UTF-8 | INS

Figure 56: Most UI related methods can be substituted with empty or single-line body.

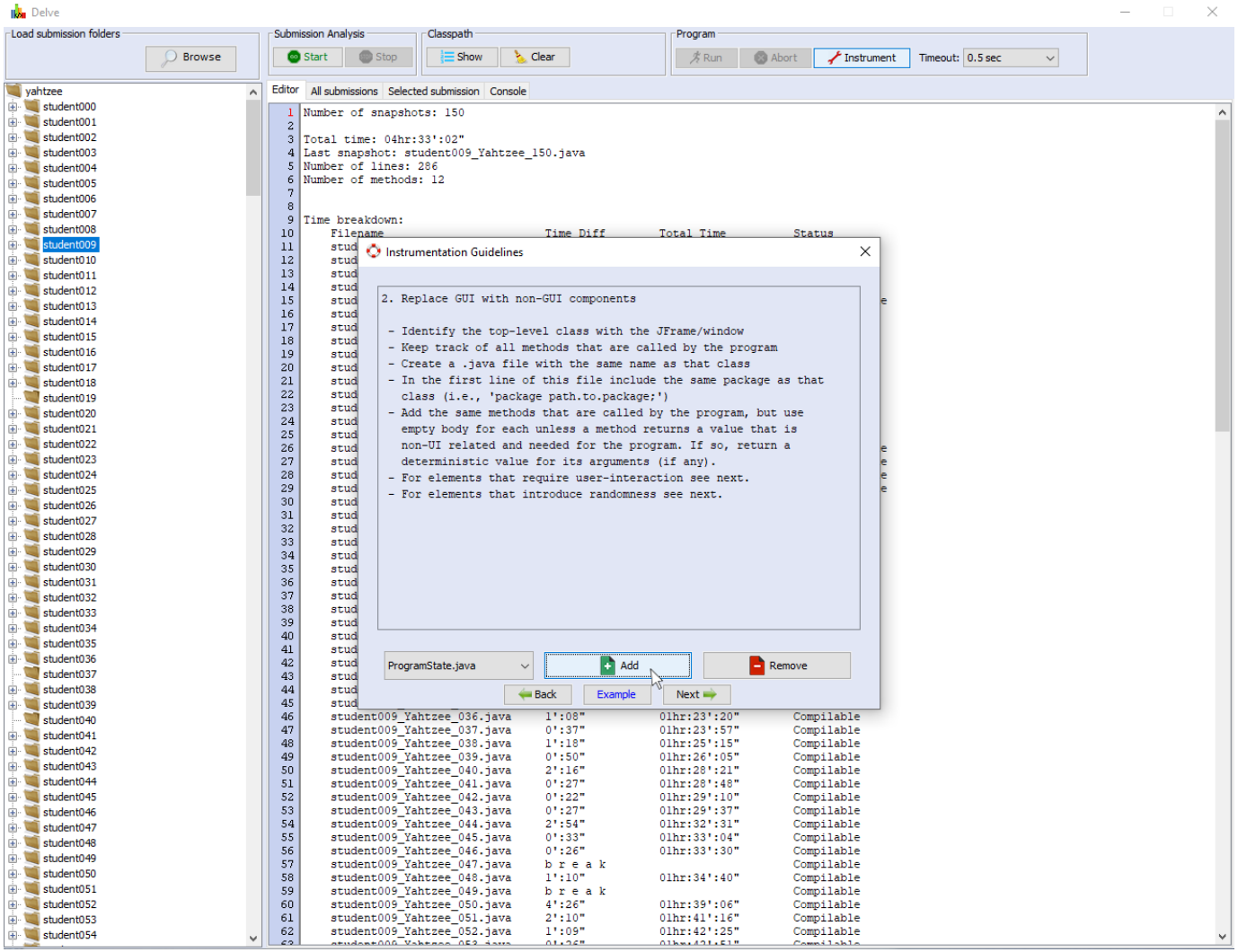

Figure 57: Click the *Add* button to locate a second file to substitute another UI module.

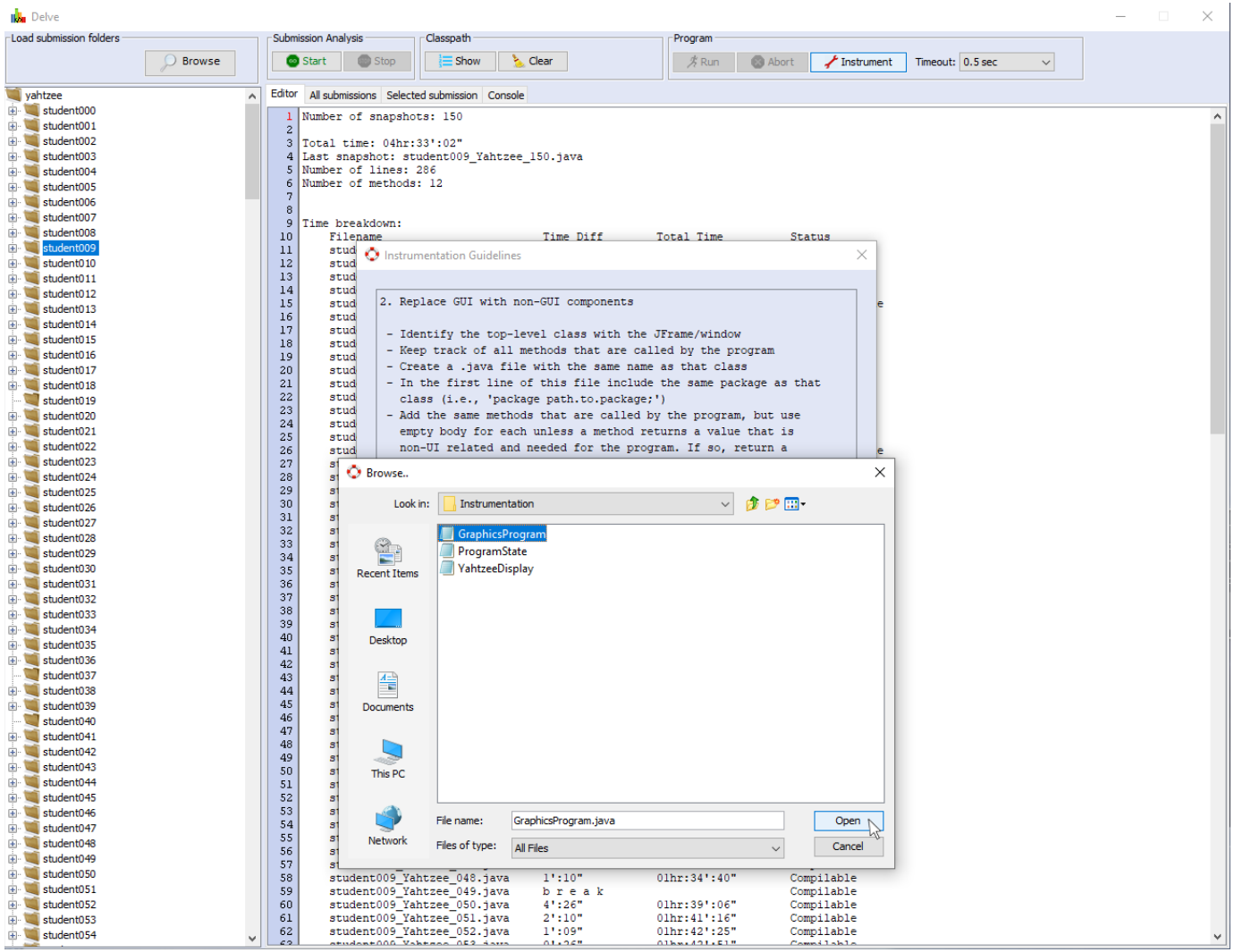

Figure 58: The user can now locate the file with the module that replaces the second UI component.

Replacing a UI module with a dummy to avoid delays is relatively easy. The goal is to bypass the UI behavior and either do nothing instead or try to capture the program state if it makes sense to do so upon a UI method call.

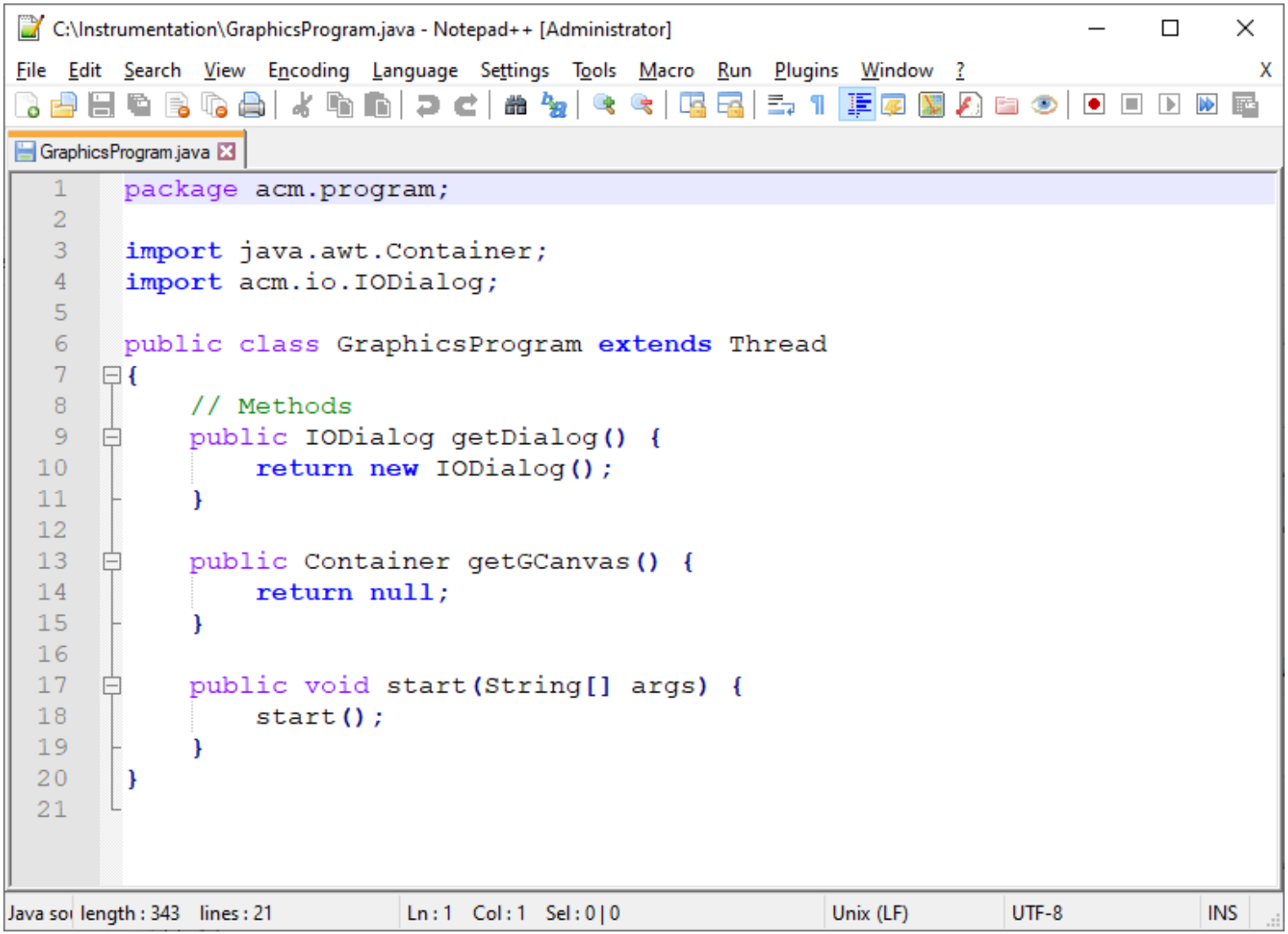

Figure 59: The UI implementation is usually simple. The goal is to bypass the UI behavior and either do nothing or try to capture the program state.

At this point we have completed the second step and are ready to move to the next step.

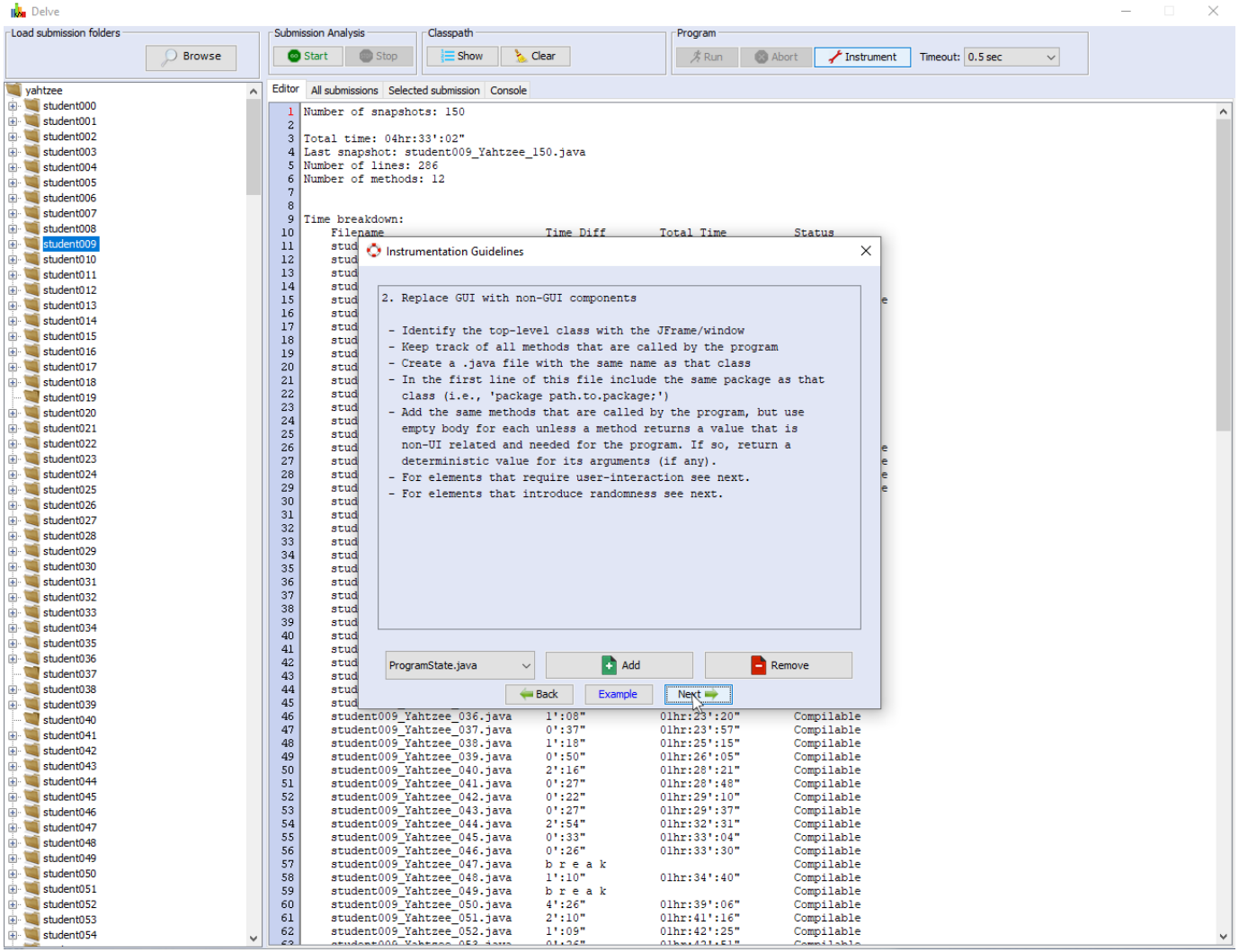

Figure 60: The second step is complete. Ready to move on to the third step.

7.4 Step 3: Get rid of user-interaction code

The third step is required if the program stalls waiting for the user to take an action (e.g., type something).

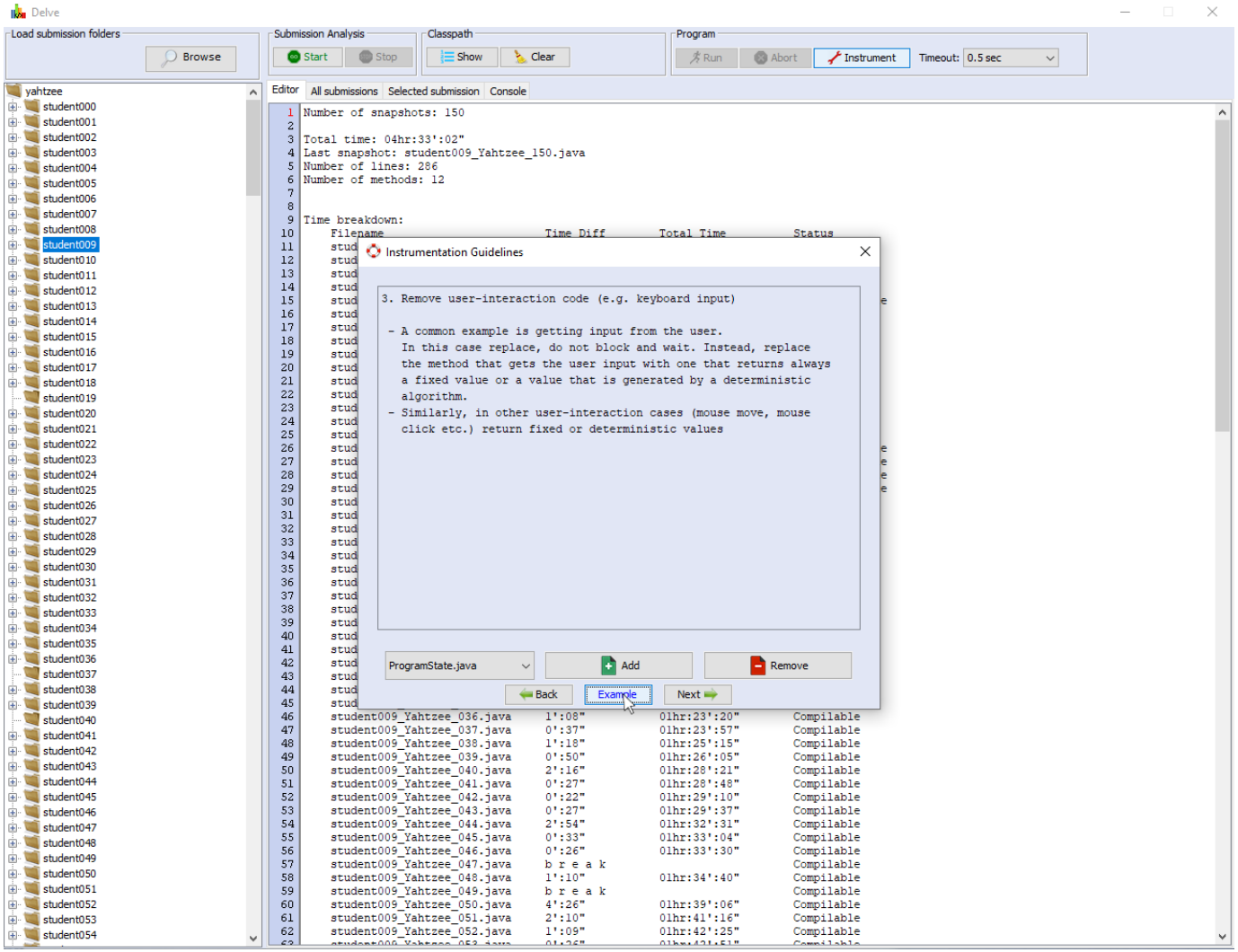

Figure 61: If there is user-interaction that stalls the execution, it should be bypassed.

Again, Delve offers an example to illustrate how this works in practice.

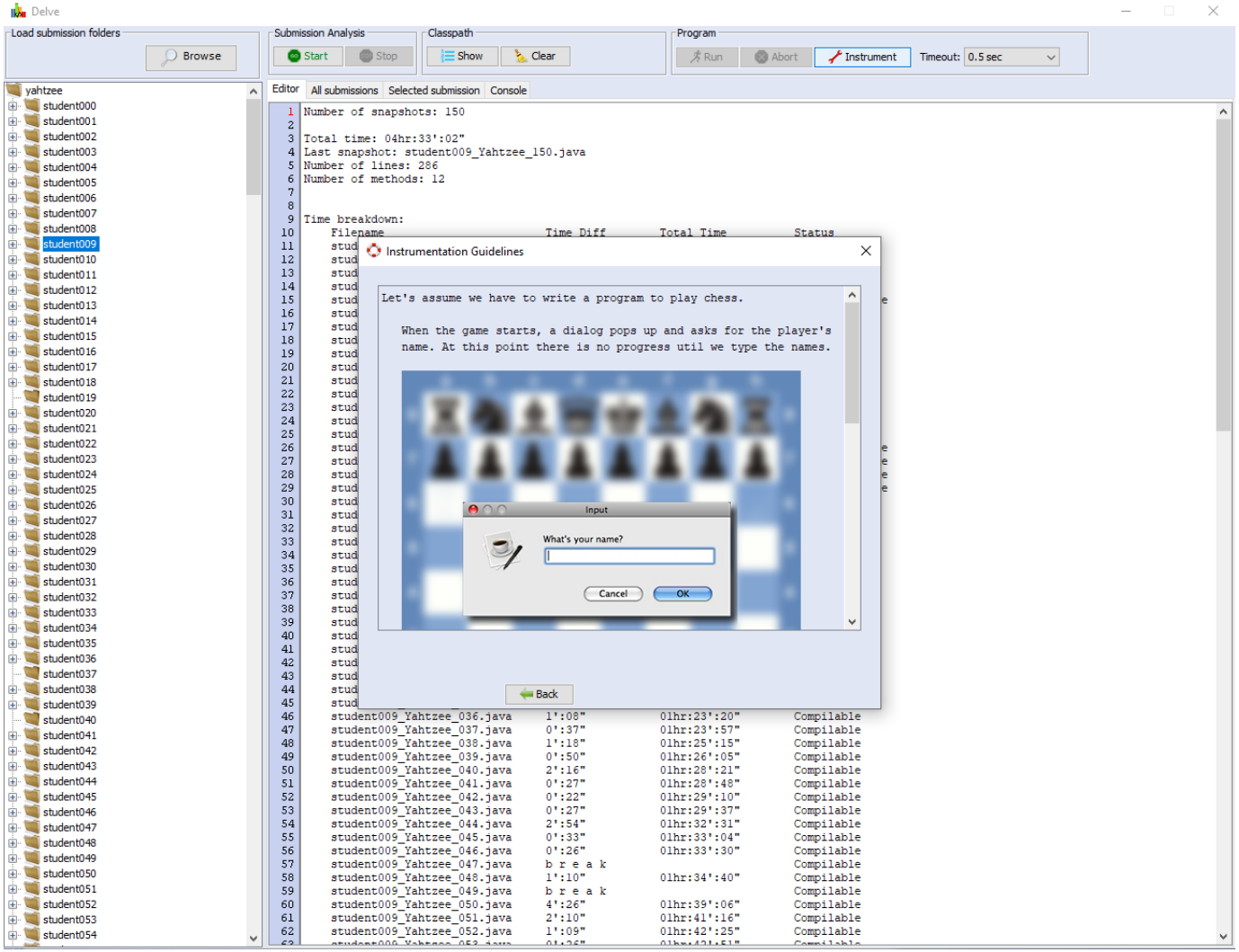

Figure 62: An example that illustrates how to bypass user-interaction that may stall the program execution.

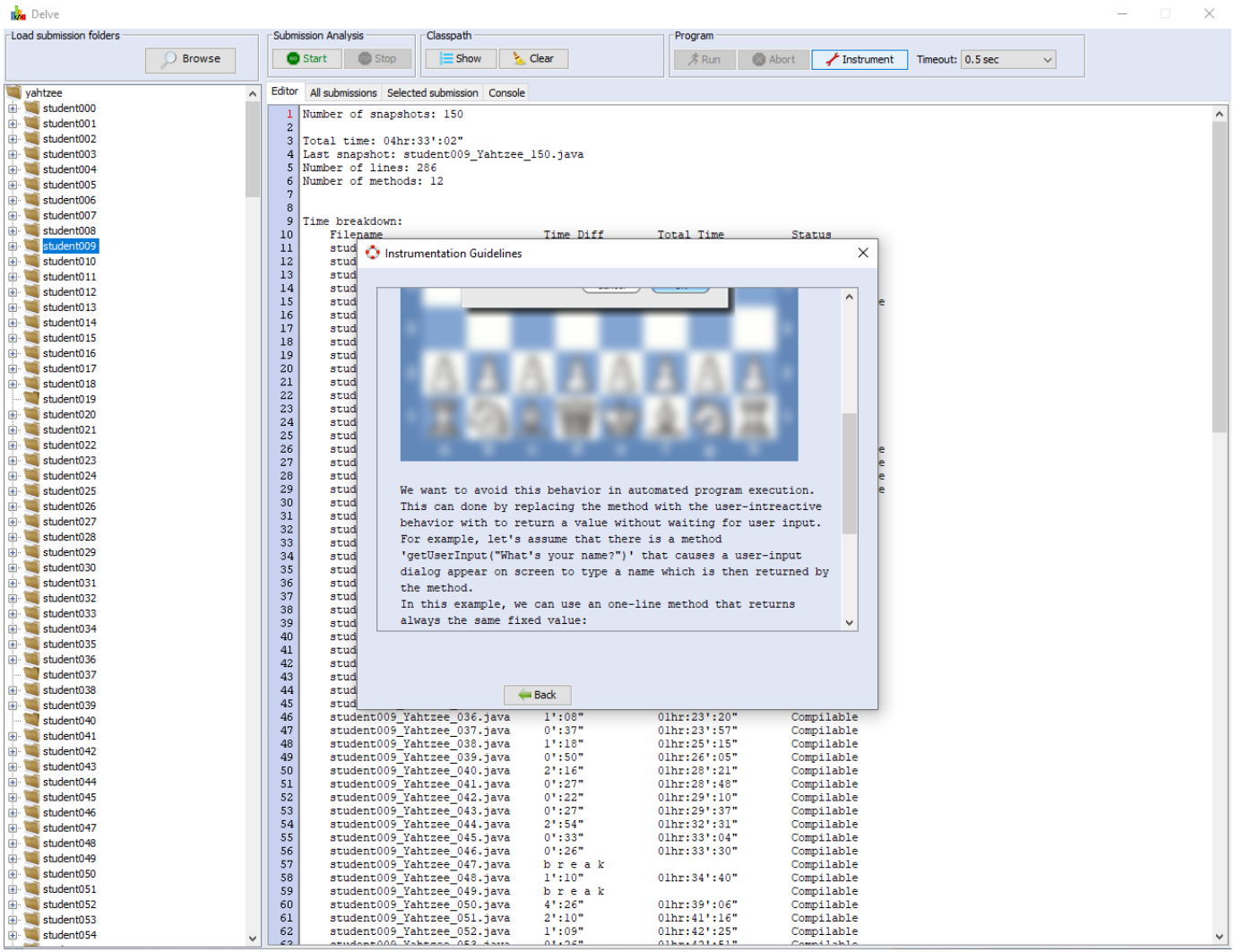

Figure 63: An example that illustrates how to bypass user-interaction that may stall the program execution.

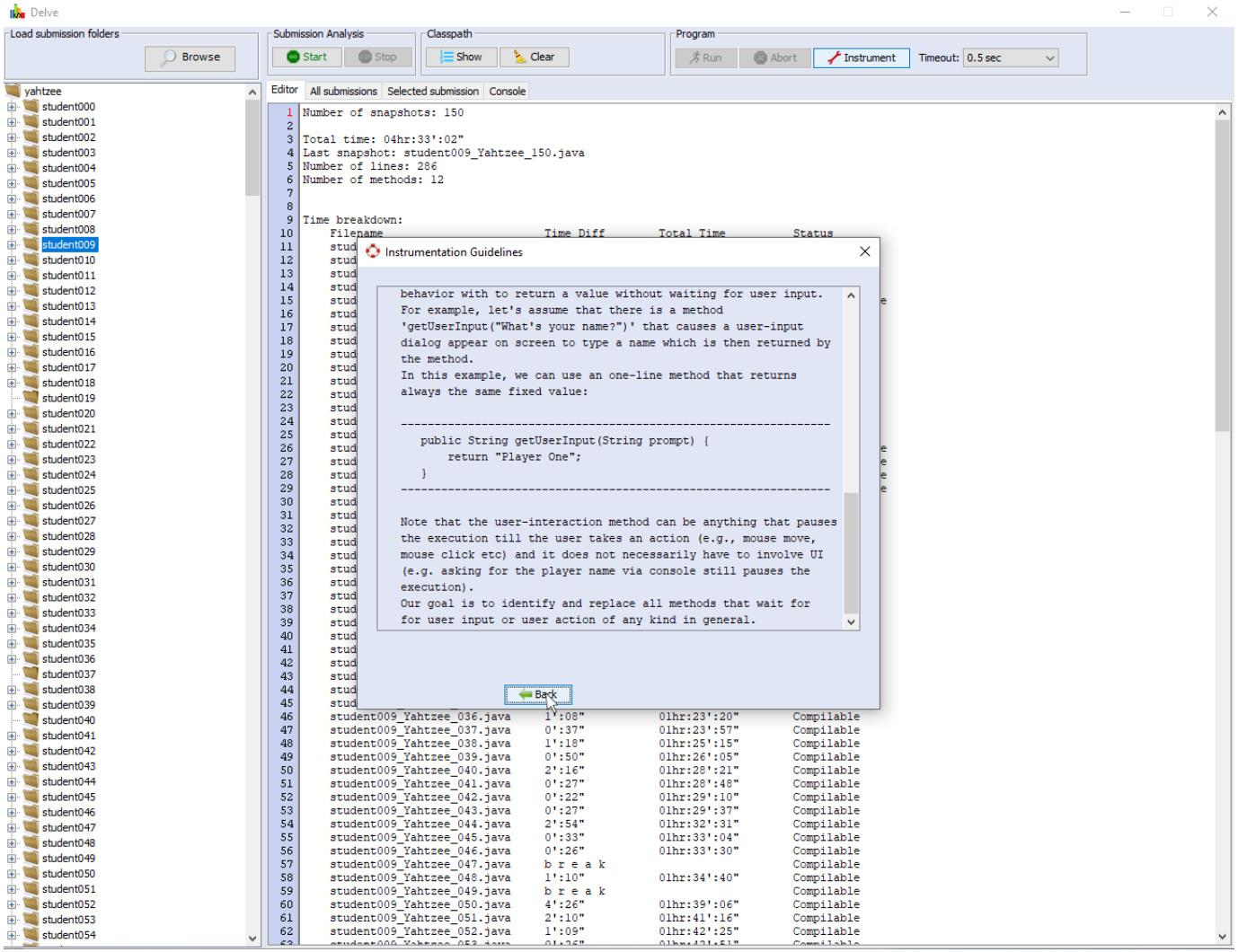

Figure 64: An example that illustrates how to bypass user-interaction that may stall the program execution.

To add the code to bypass user-interaction that stalls execution, the user can click the **Add** button

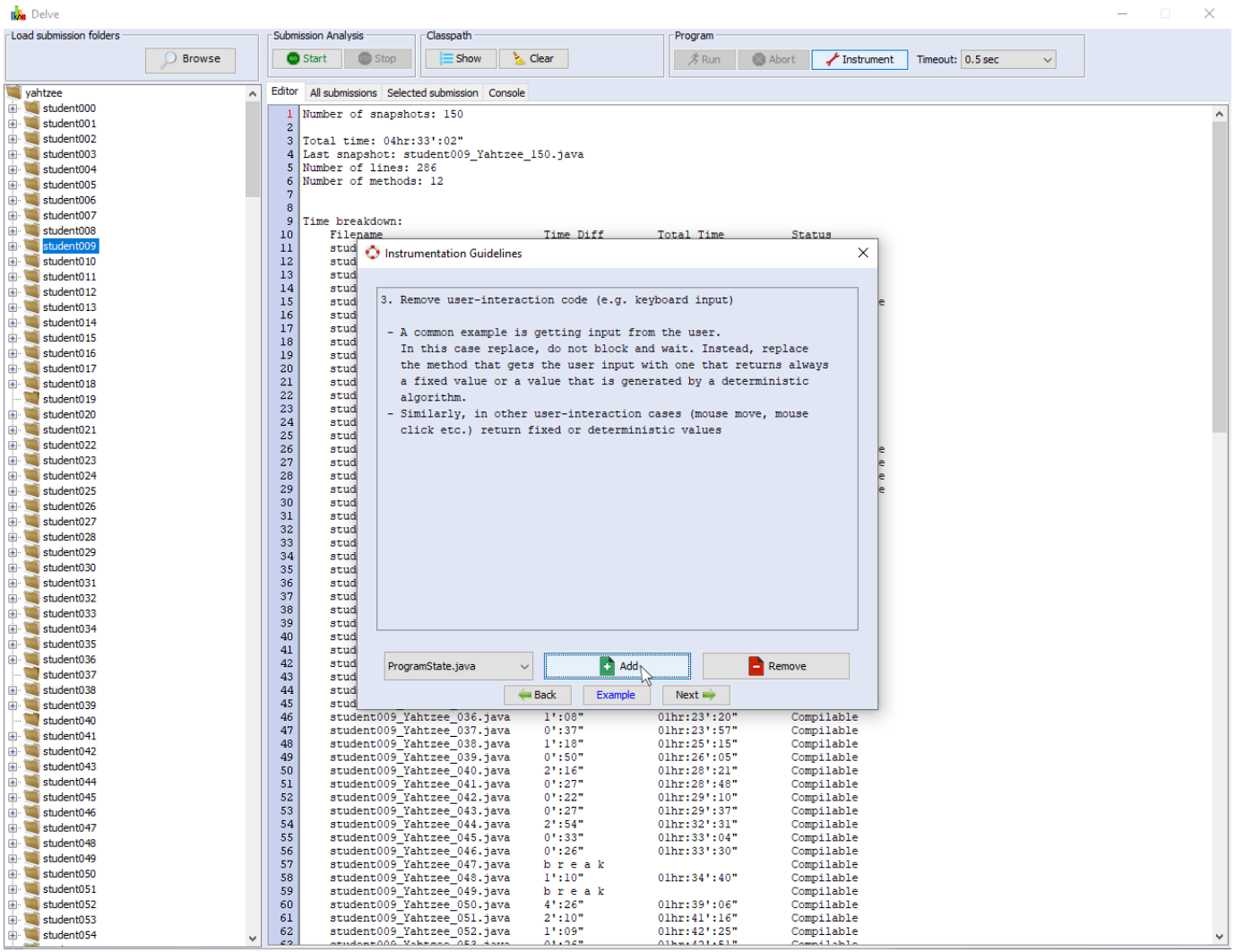

Figure 65: Click the *Add* button to locate the file with the code to bypass any user-interaction that may stall the execution.

and then locate the file(s) to bypass user-interaction.

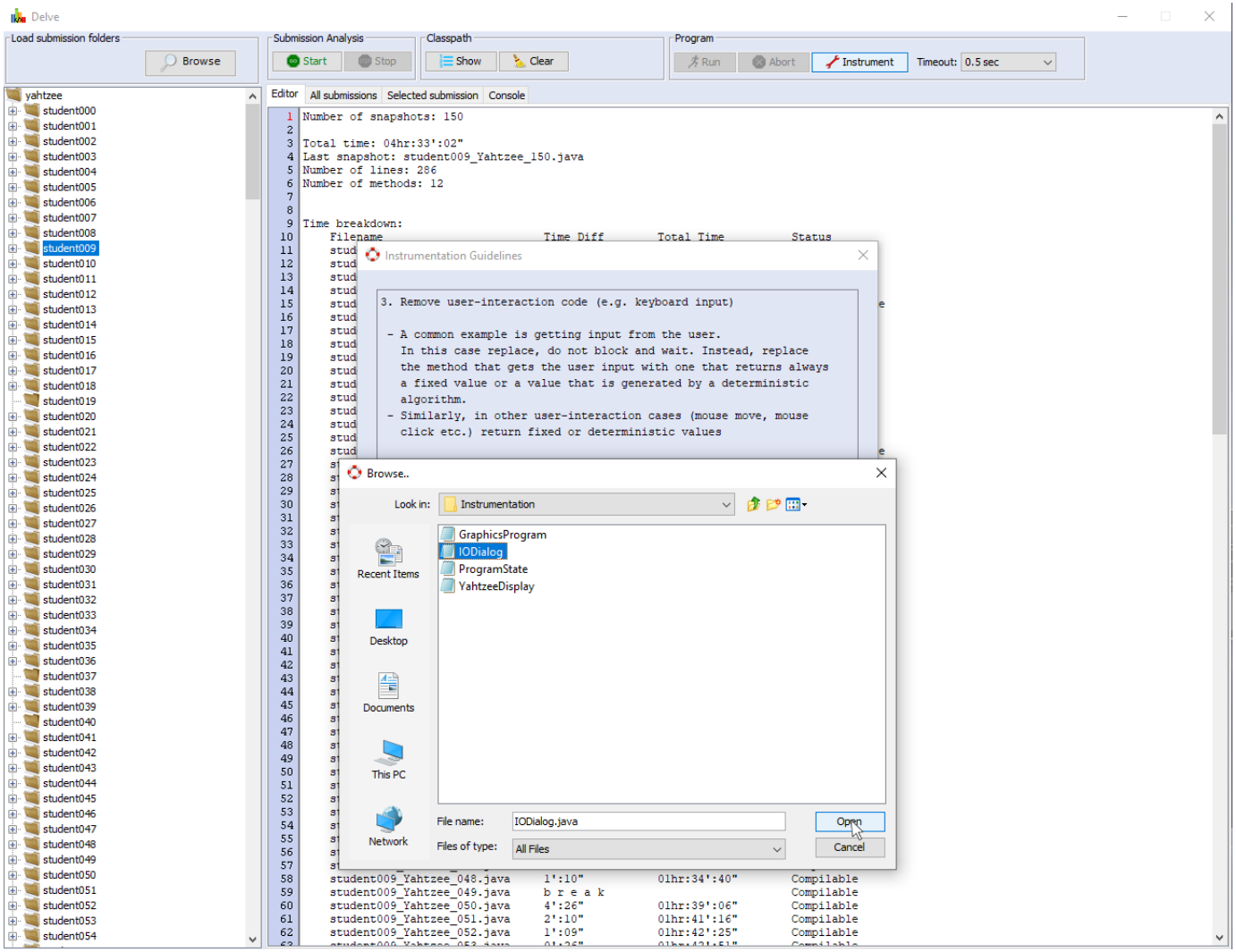

Figure 66: The user can now locate the file with the code to bypass any user-interaction.

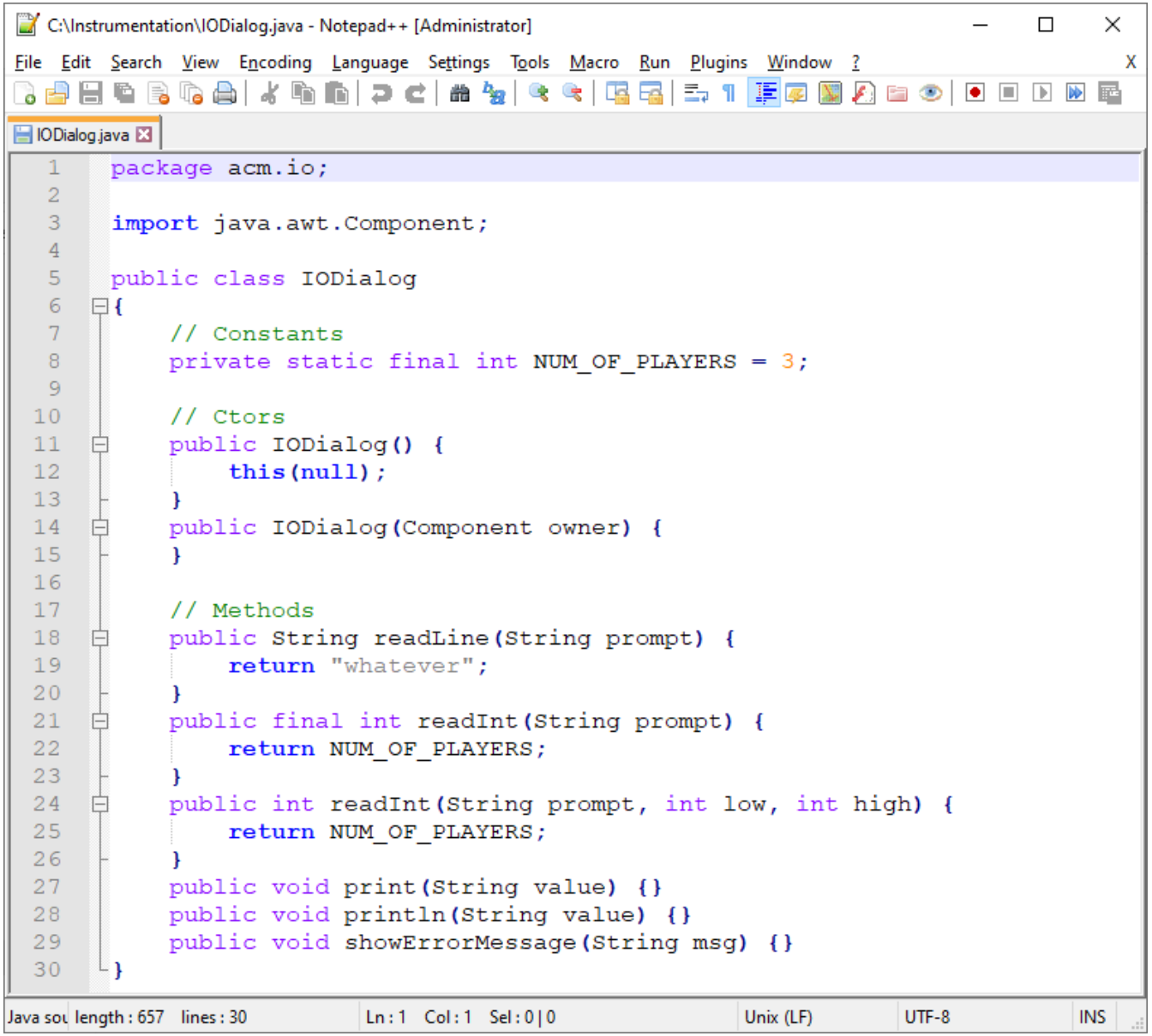

Figure 67: Replacing the user-interaction methods is straightforward. Occasionally, it may require some logic to return the proper value.

At this point we have completed the third step and are ready to move to the next step.

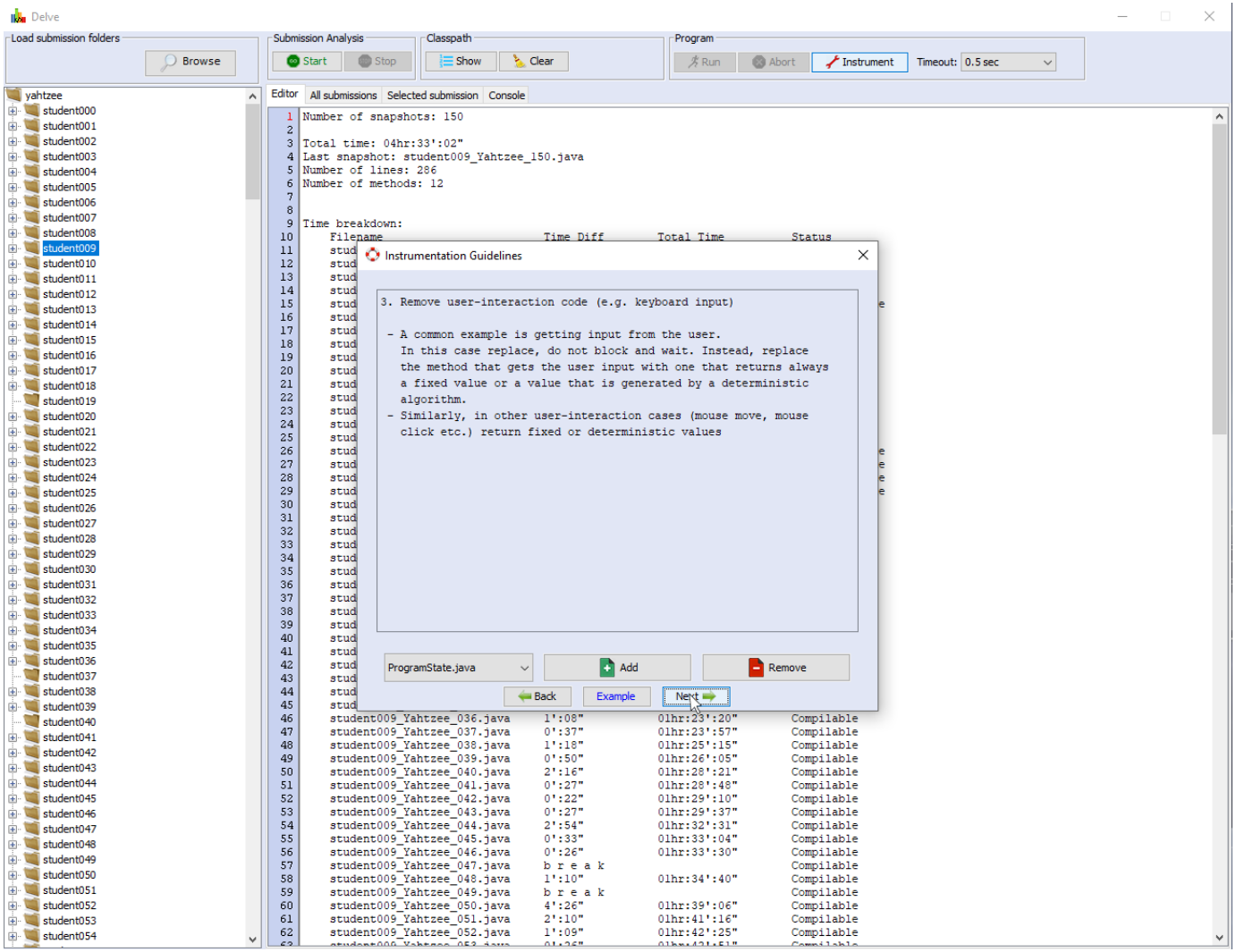

Figure 68: The third step is complete. Ready to move on to the fourth step.

7.5 Step 4: Replace random with deterministic behavior

The fourth step is required if the program has an element of randomness and does not produce always the same result.

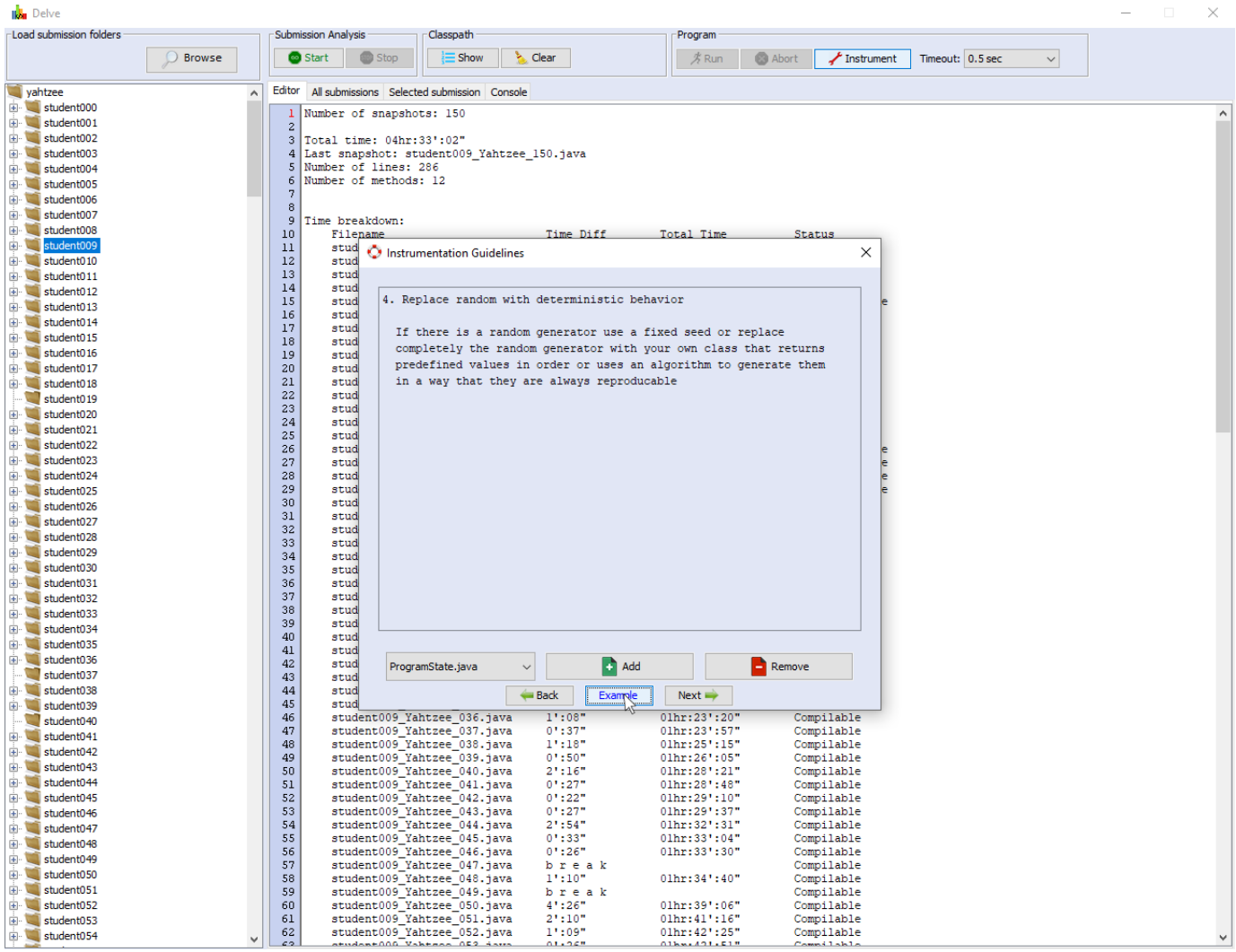

Figure 69: If there is an element of randomness, it should be replaced by deterministic behavior.

Again, Delve offers an example to illustrate how this works in practice.

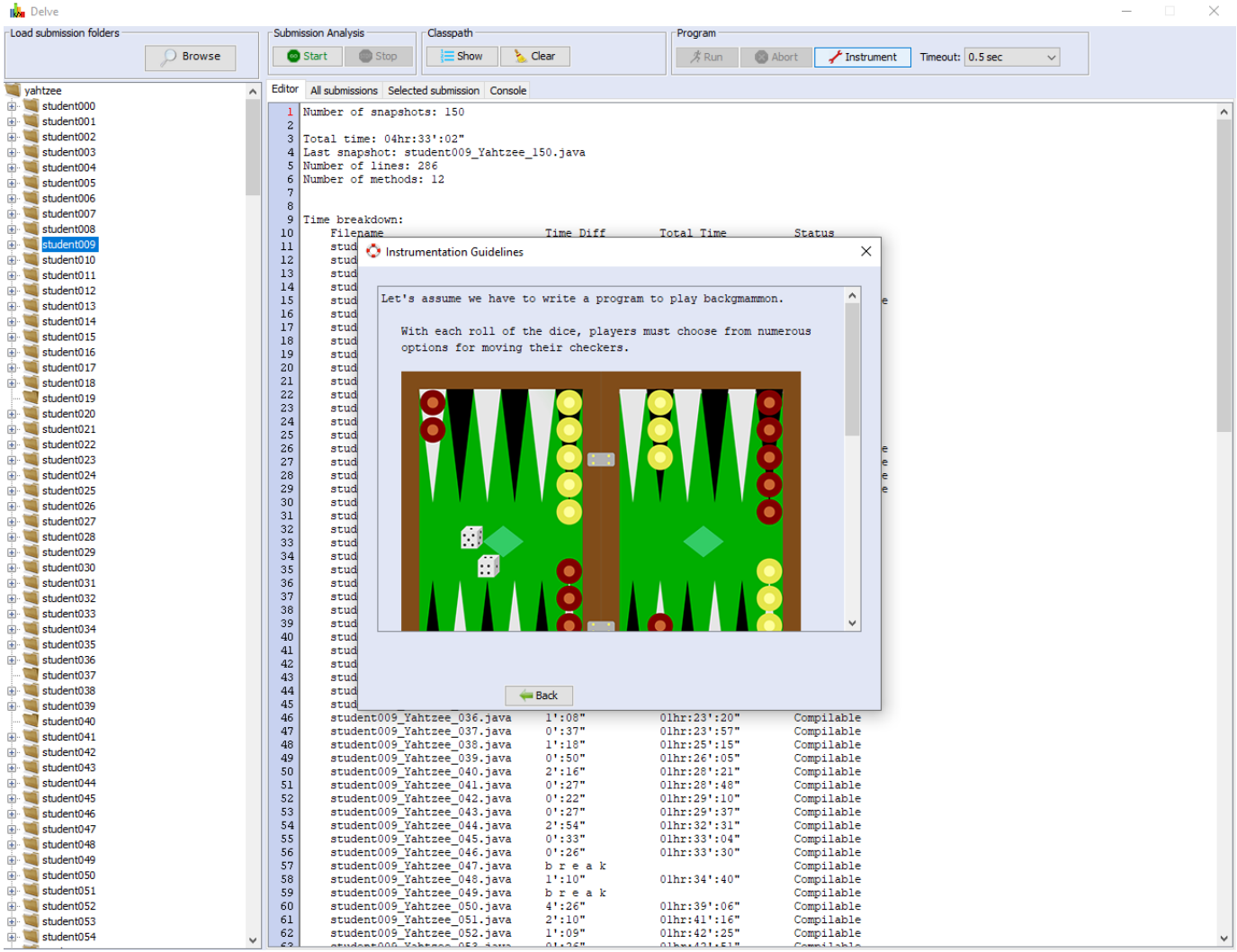

Figure 70: An example that illustrates how to remove elements of randomness from the program execution.

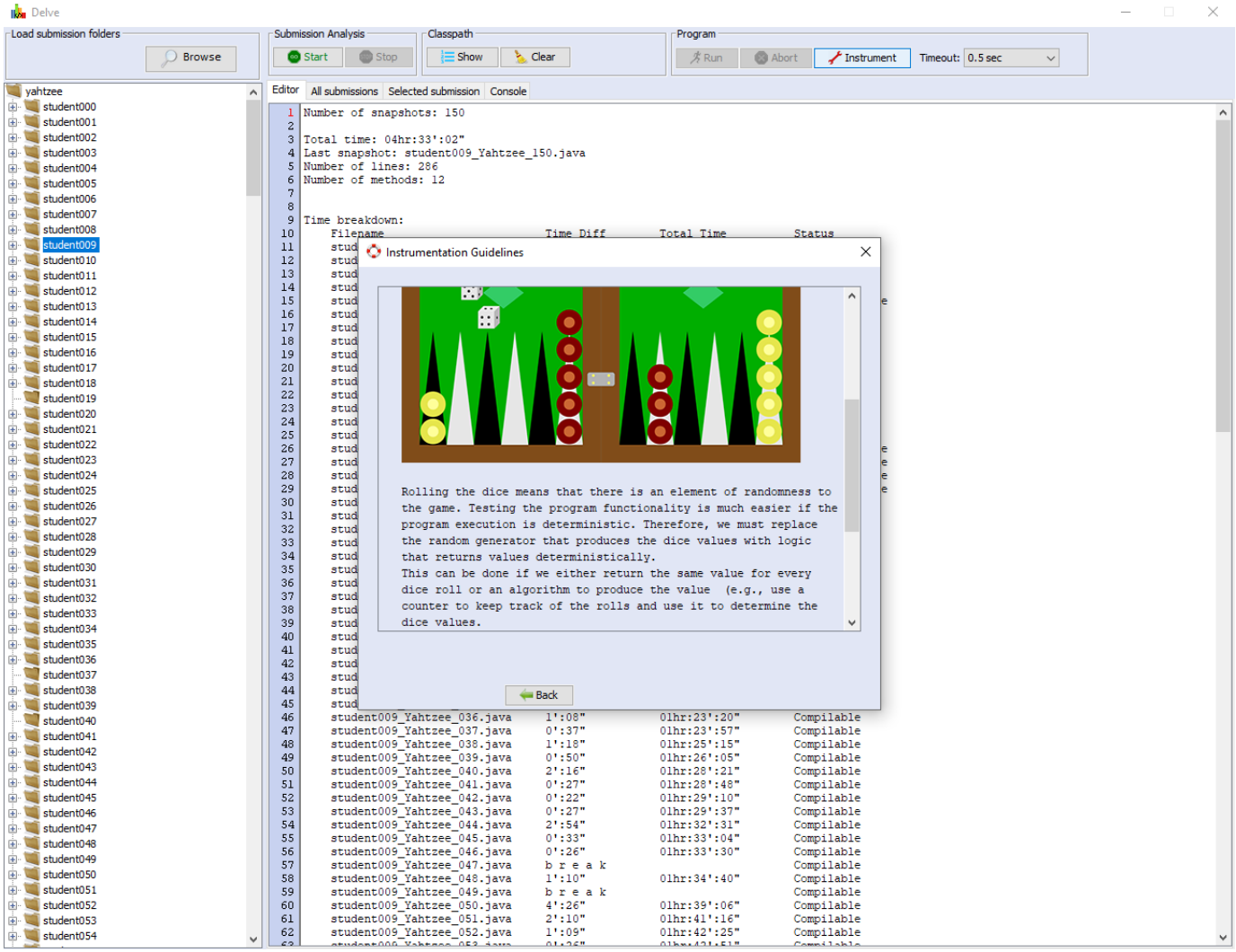

Figure 71: An example that illustrates how to remove elements of randomness from the program execution.

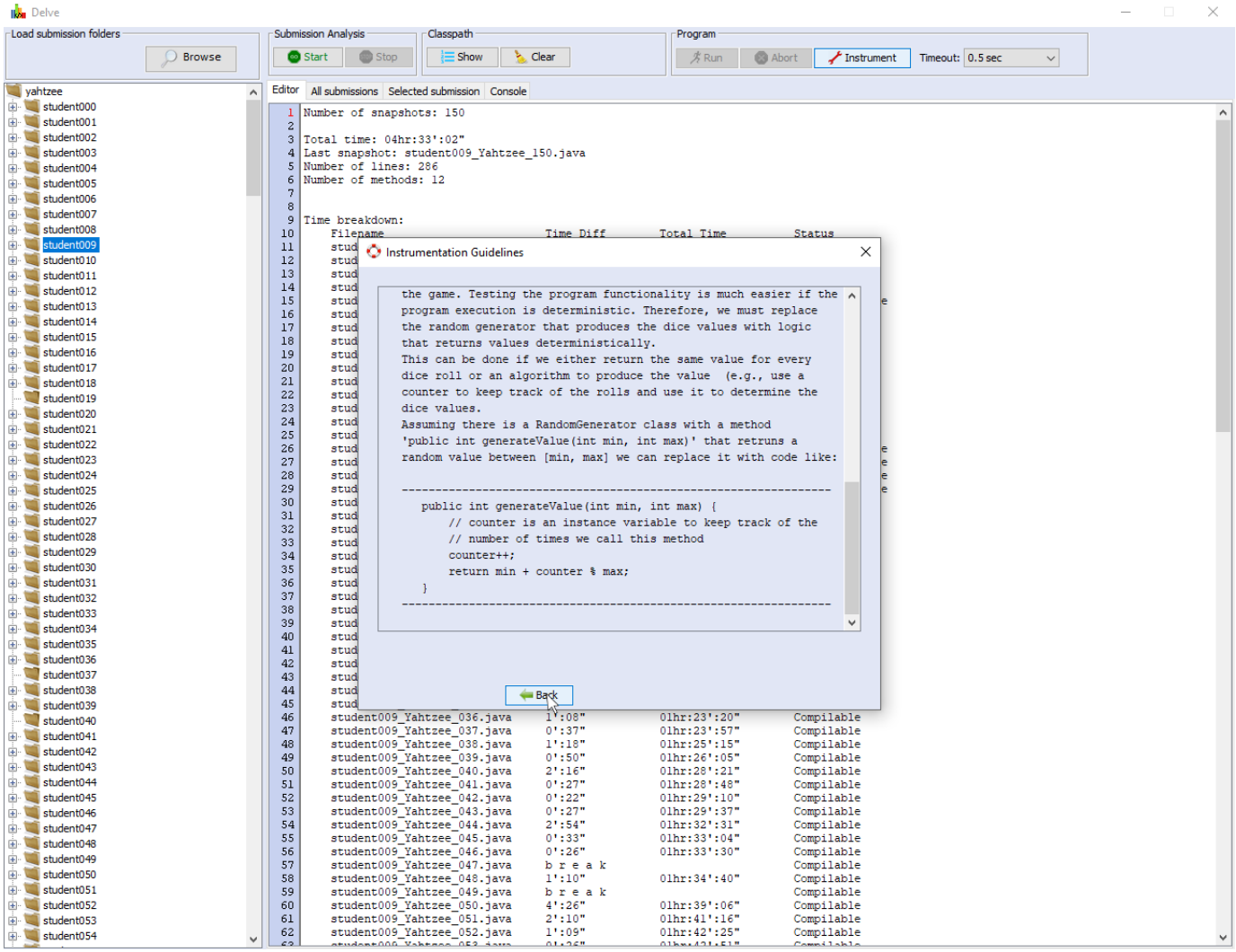

Figure 72: An example that illustrates how to remove elements of randomness from the program execution.

To add the code to replace random with deterministic behavior, the user can click the **Add** button

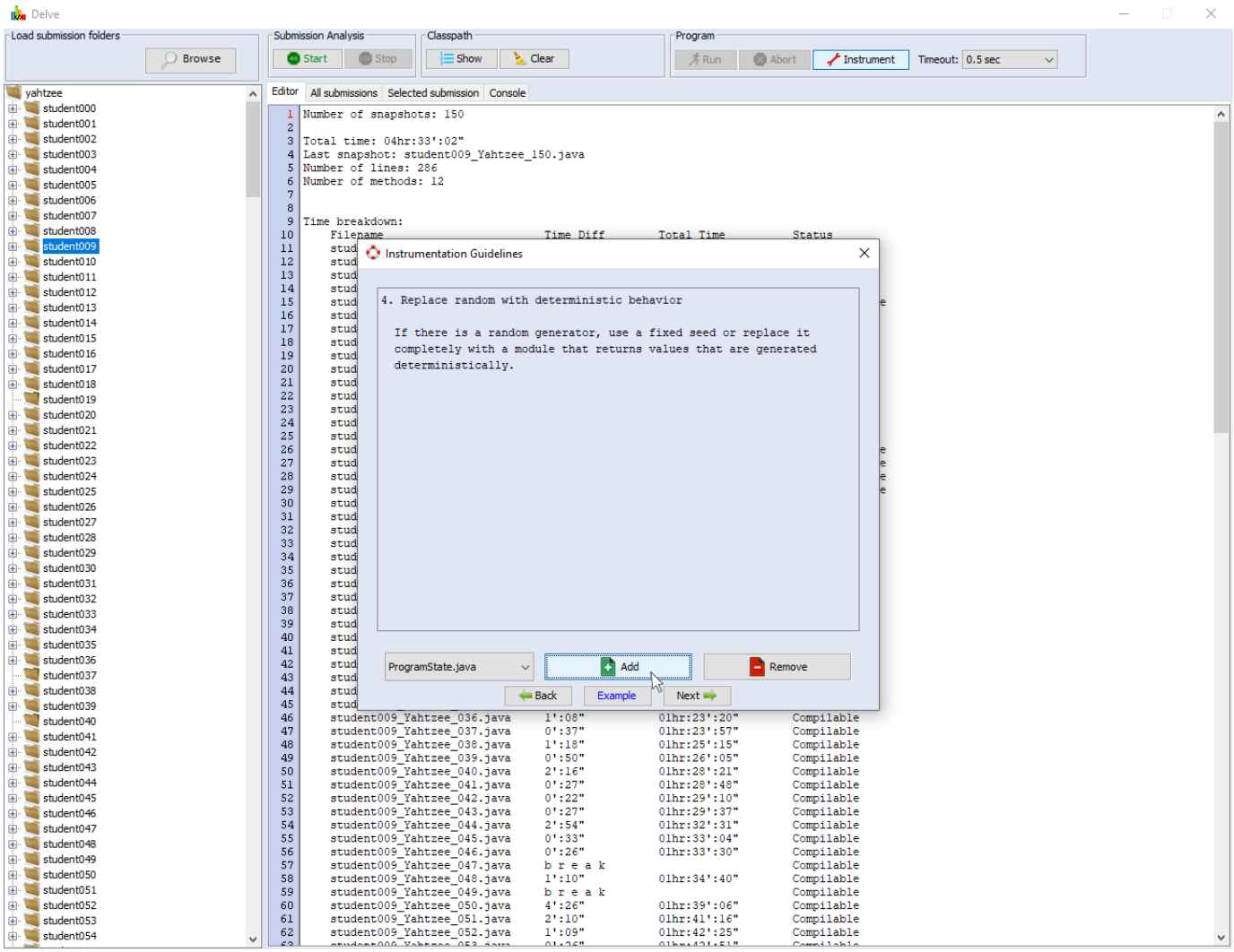

Figure 73: Click the *Add* button to locate the file with the code to replace random with deterministic behavior.

and then locate the file(s) to ensure deterministic behavior.

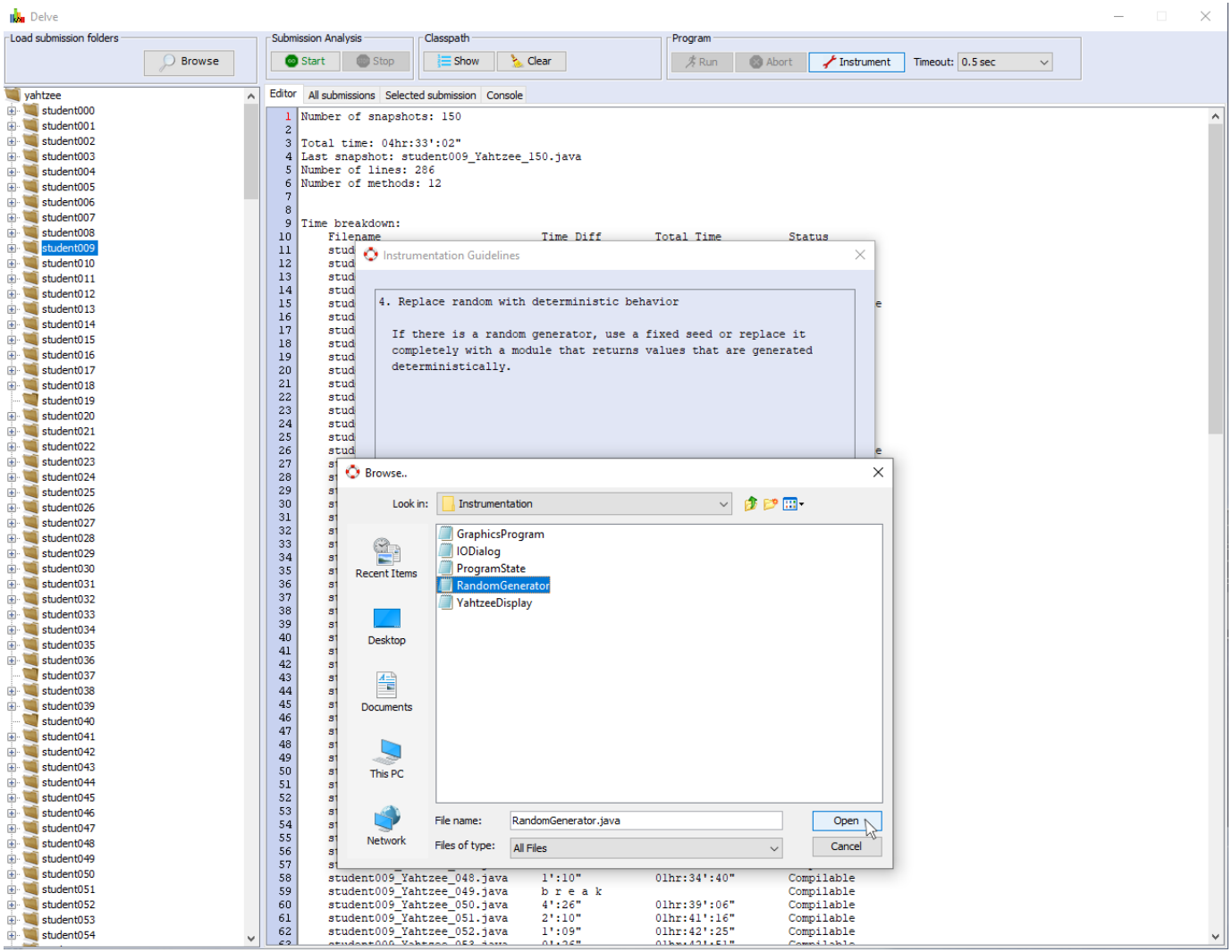

Figure 74: The user can now locate the file with the code to replace random with deterministic behavior.

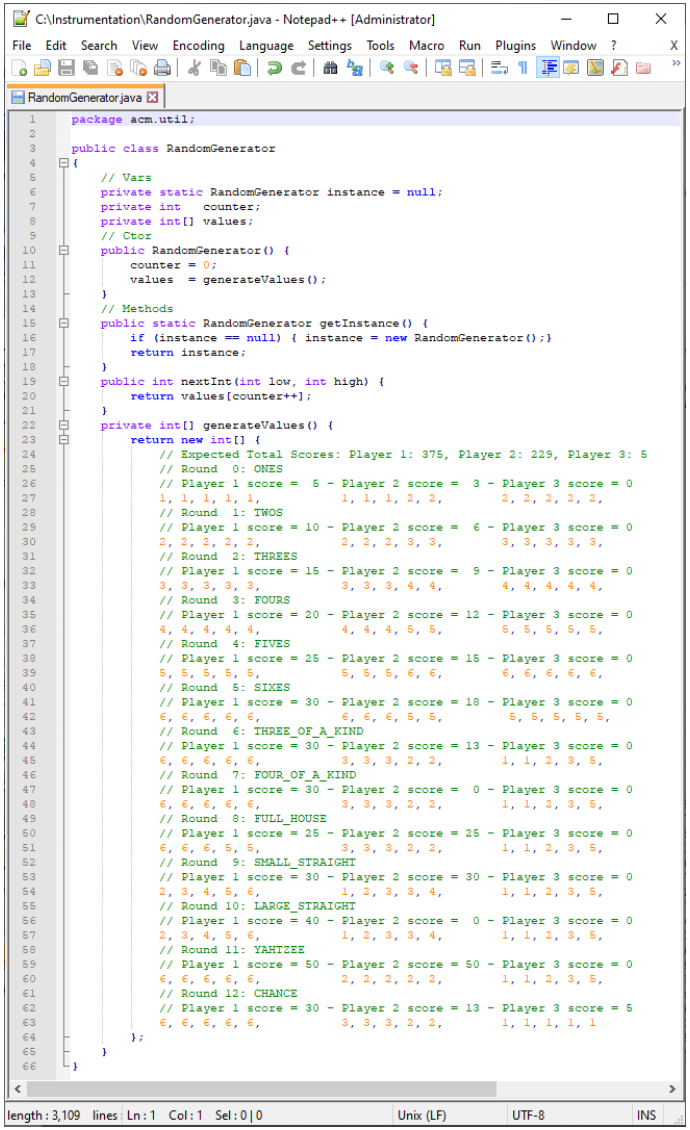

Figure 75: In this example, we want to emulate rolling the dice. Instead of producing random values, the code keeps track of the values returned so far and uses that variable (i.e., count) as index to a fixed array of predefined values.

At this point we have completed the fourth step and are ready to move to the next step.

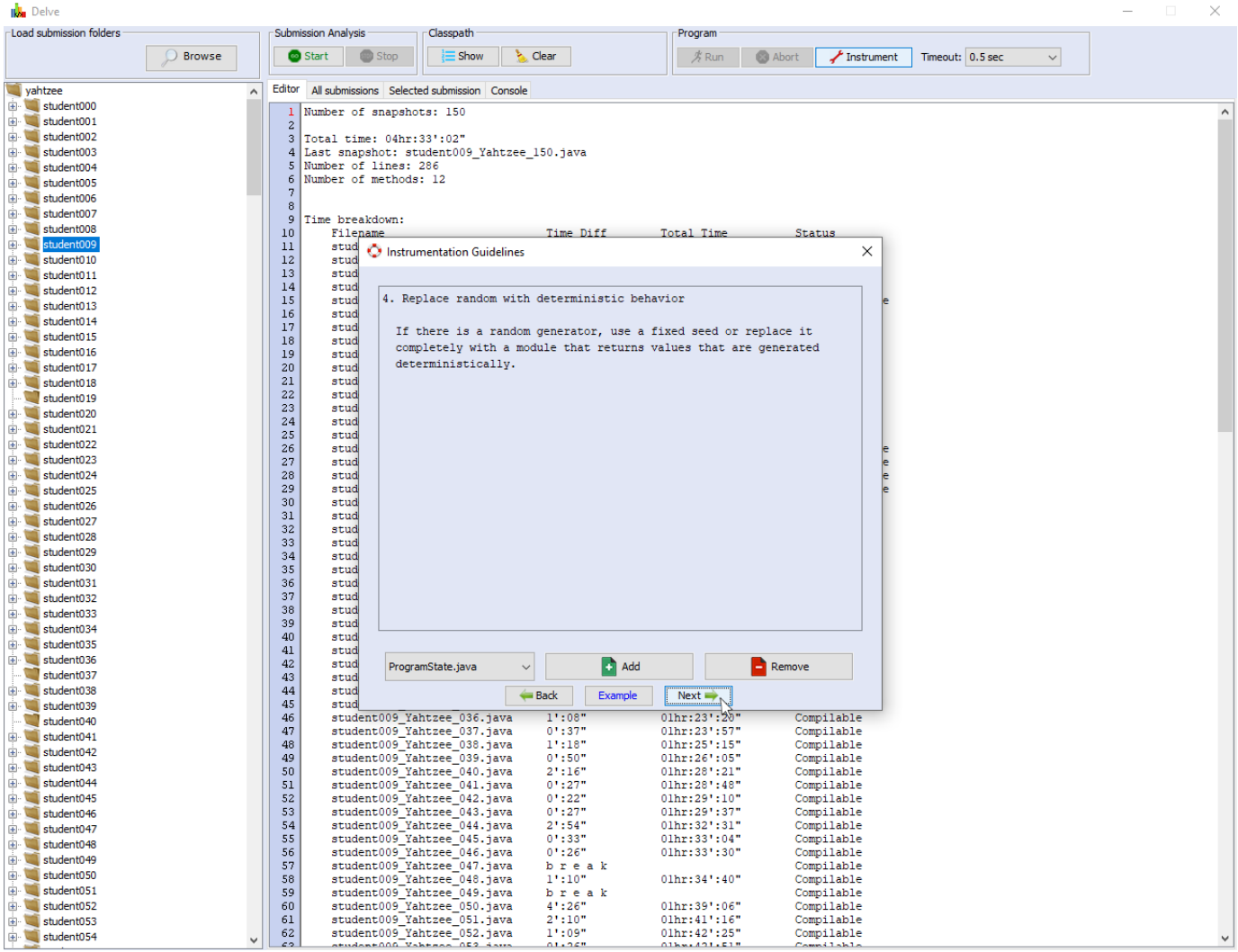

Figure 76: The fourth step is complete. Ready to move on to the fifth step.

7.6 Step 5: Save the program state

Each student program is different. If we want to instrument the code to save the state for all programs, we have to identify one or more methods that every program calls and try to capture the state at this point, if possible.

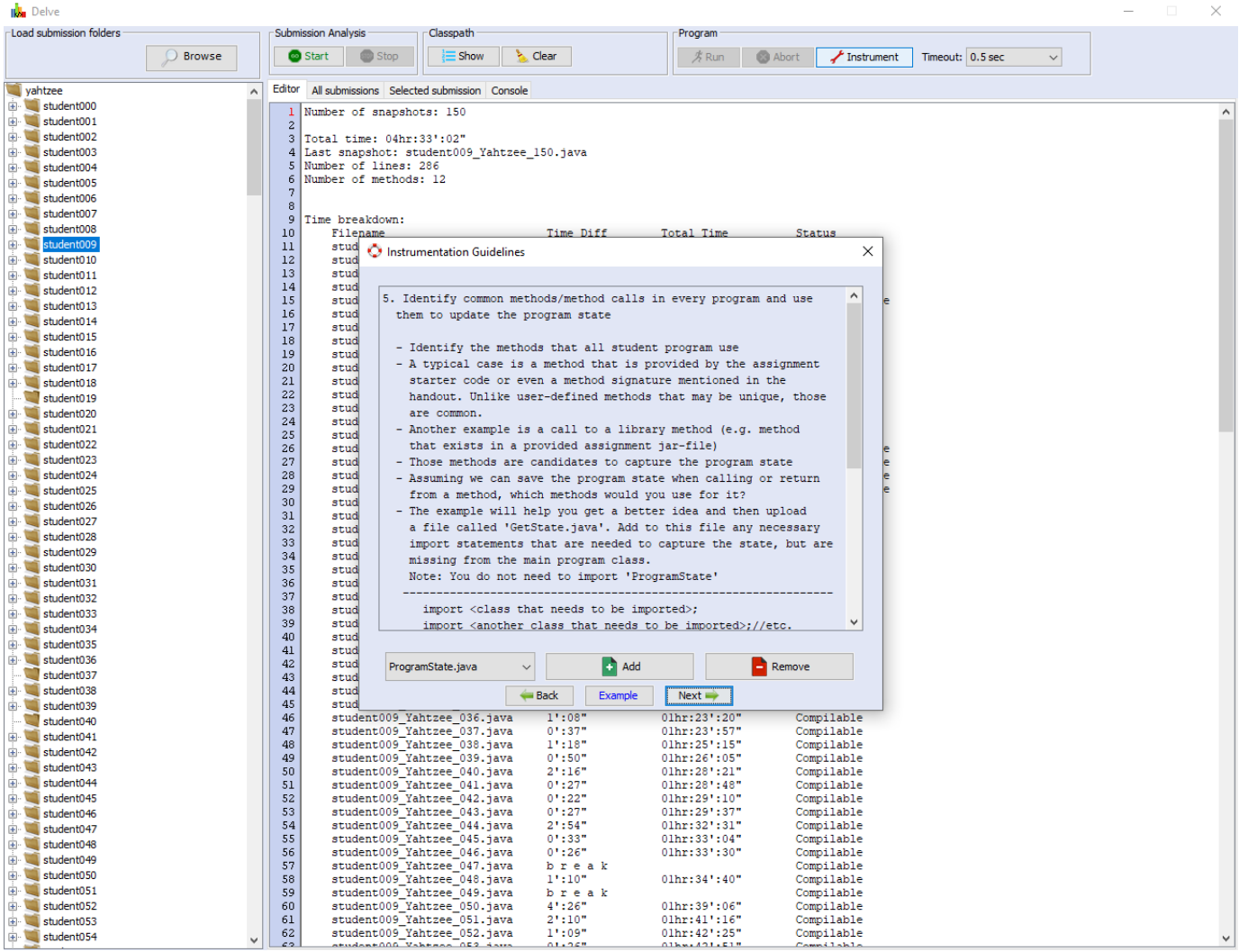

Figure 77: Delve's tips to identify common methods that can help to capture the program state.

Delve provides a few hints and lists also all the common internal program methods (i.e., methods that belong to the program that the student writes) as shown in the figure below. However, there are more places to consider. For example, each program may call a library API. Thus, candidate places to capture the state, are not just common methods in the program itself, but also external methods that all programs call.

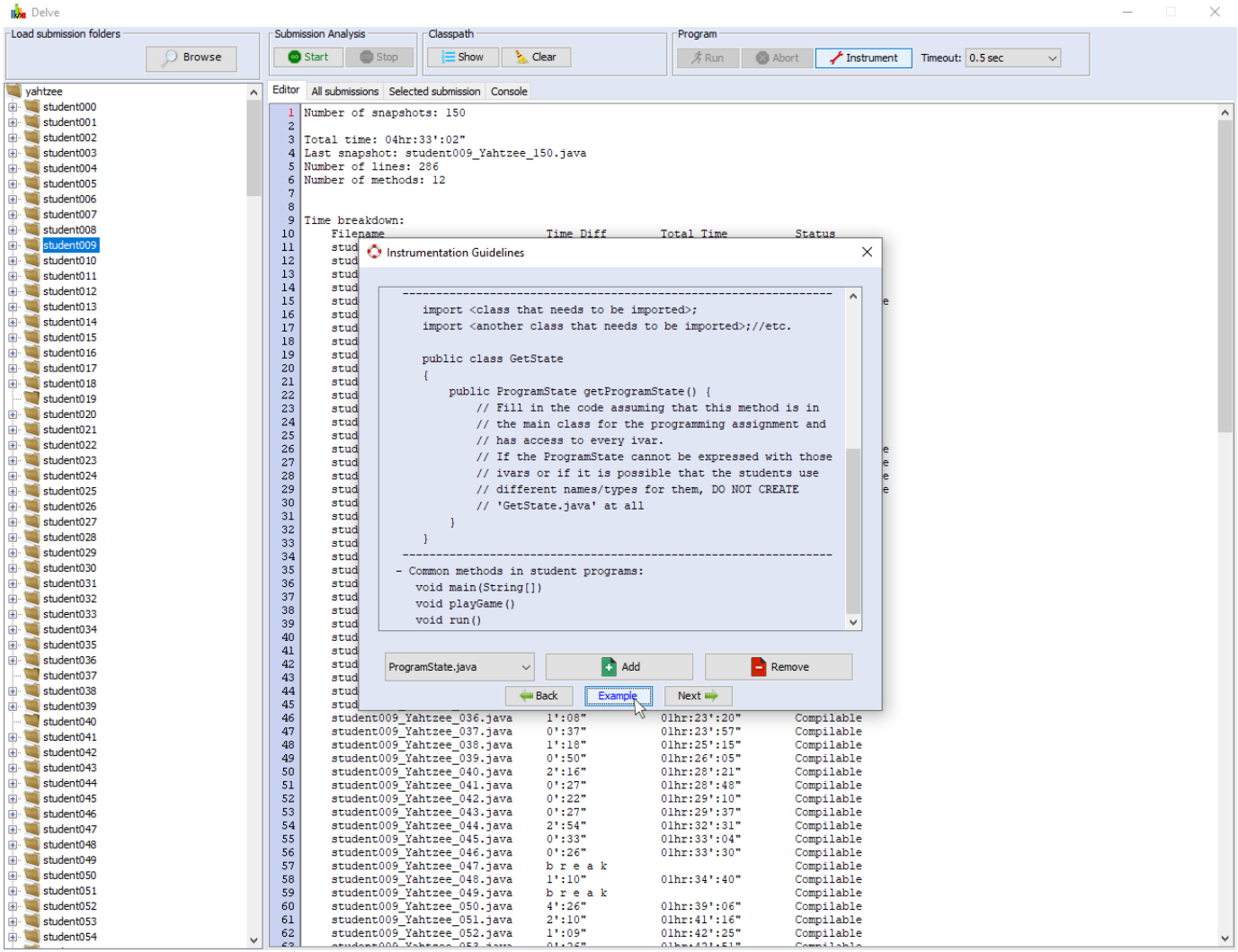

Figure 78: Delve lists the internal methods that are common in the student programs.

Again, Delve offers an example to illustrate how this works in practice.

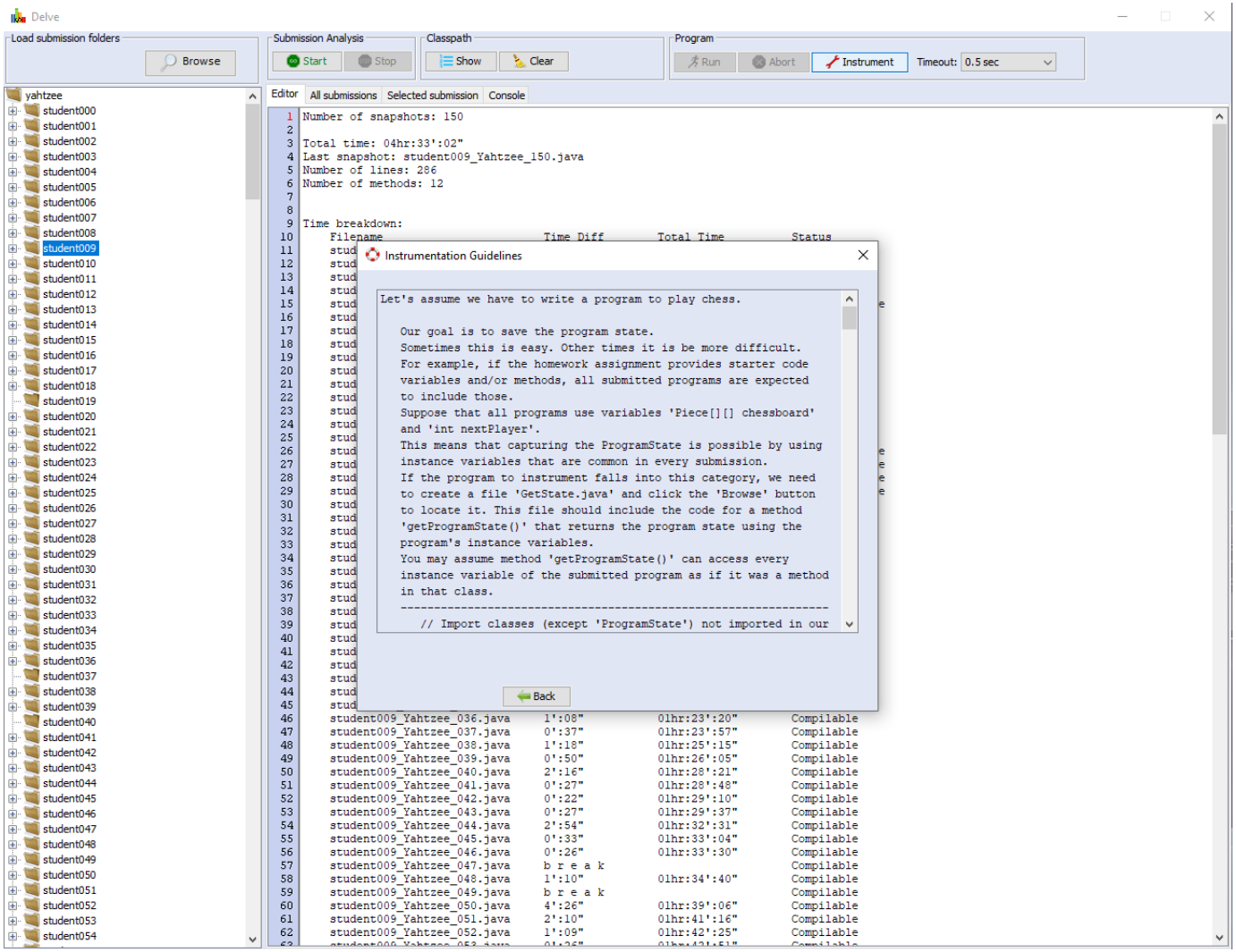

Figure 79: An example to show how to save the program state in cases when it is not so obvious.

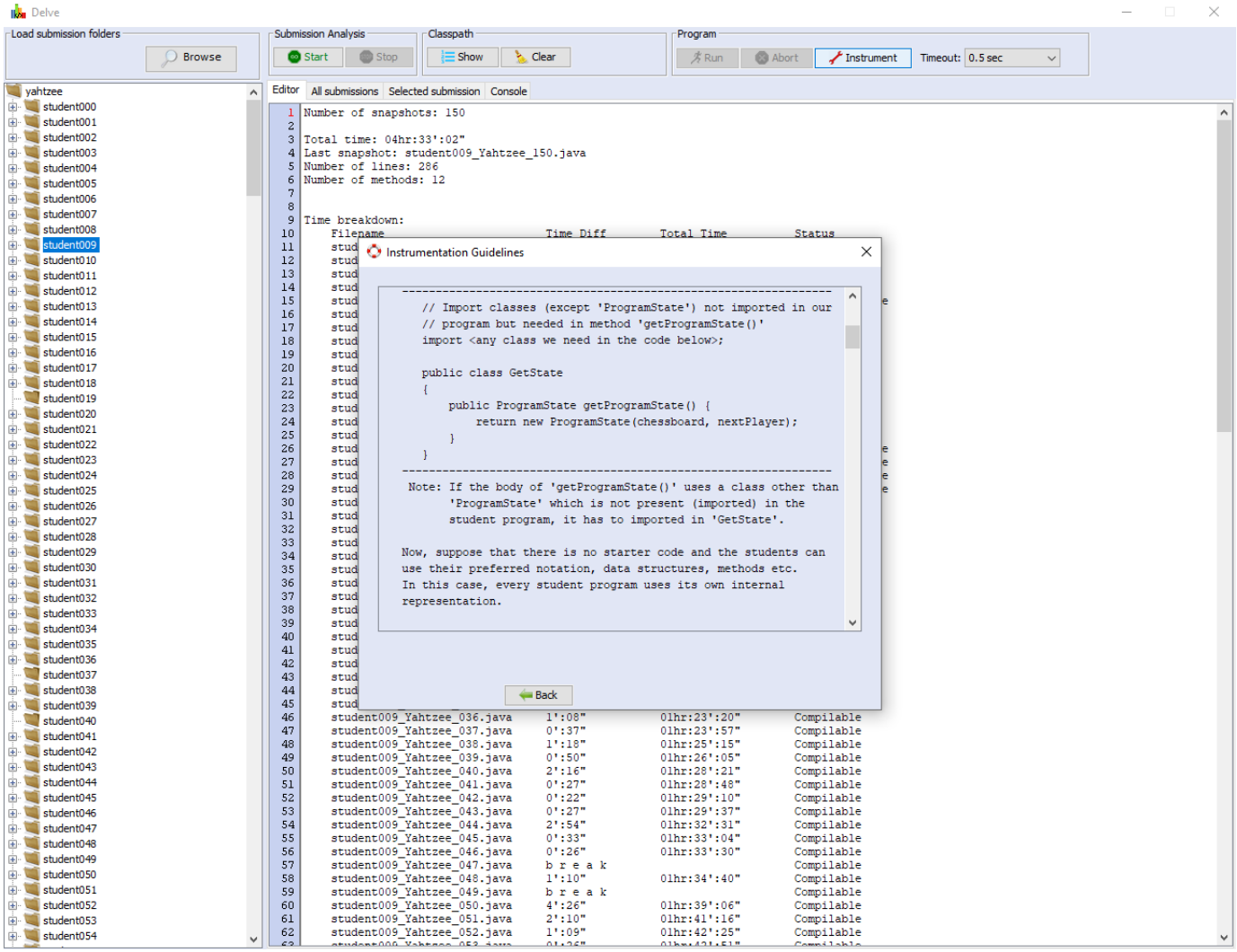

Figure 80: An example to show how to save the program state in cases when it is not so obvious.

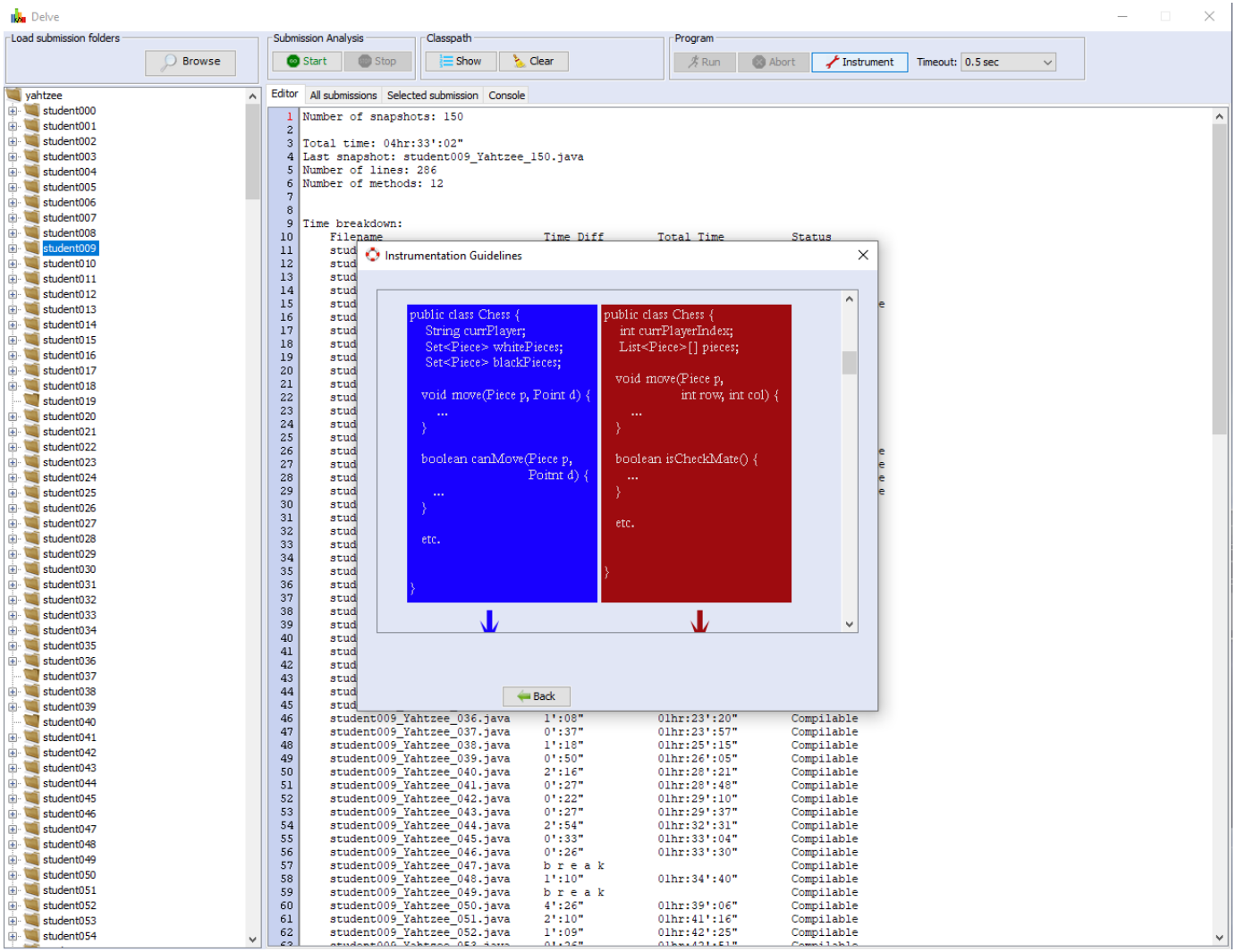

Figure 81: An example to show how to save the program state in cases when it is not so obvious.

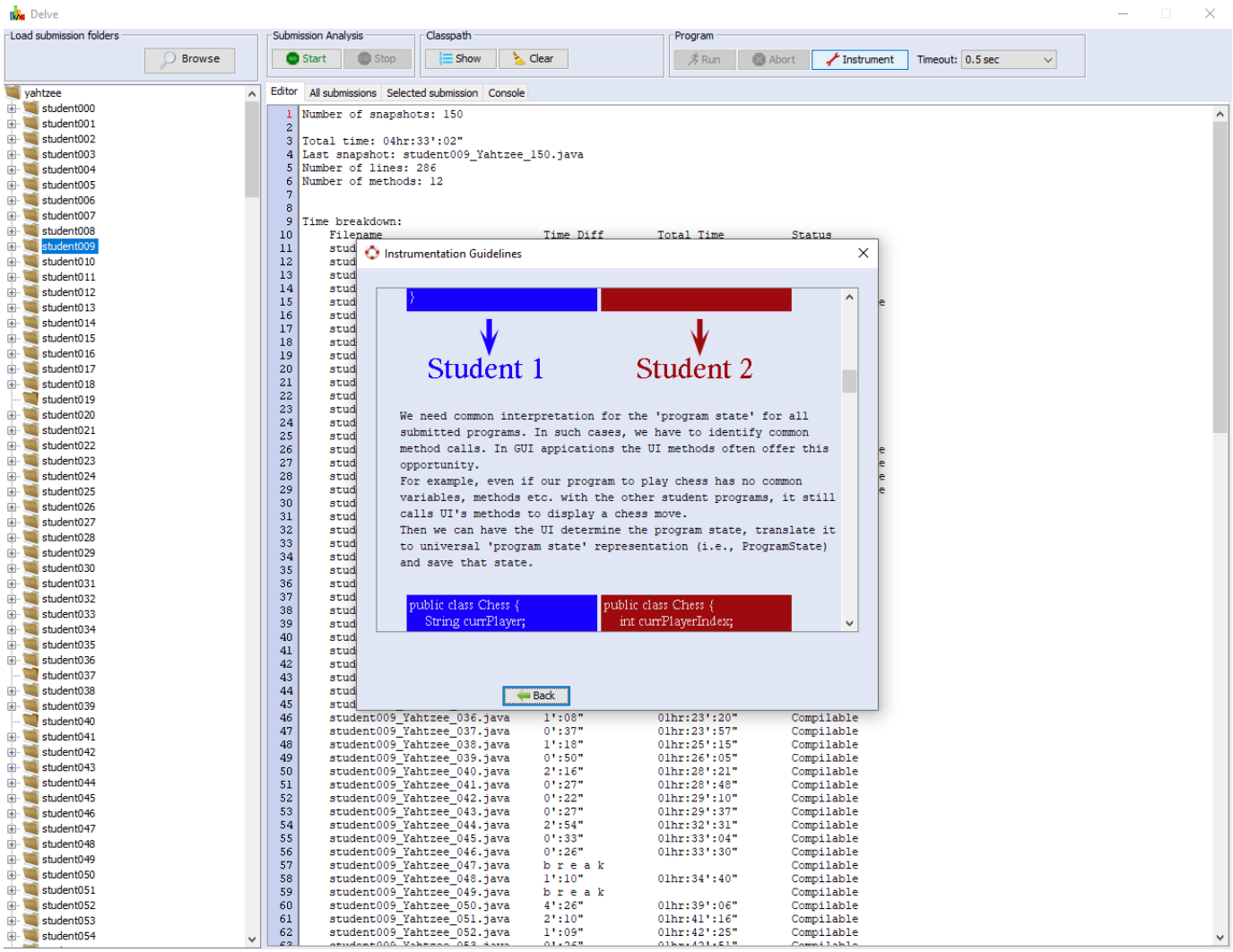

Figure 82: An example to show how to save the program state in cases when it is not so obvious.

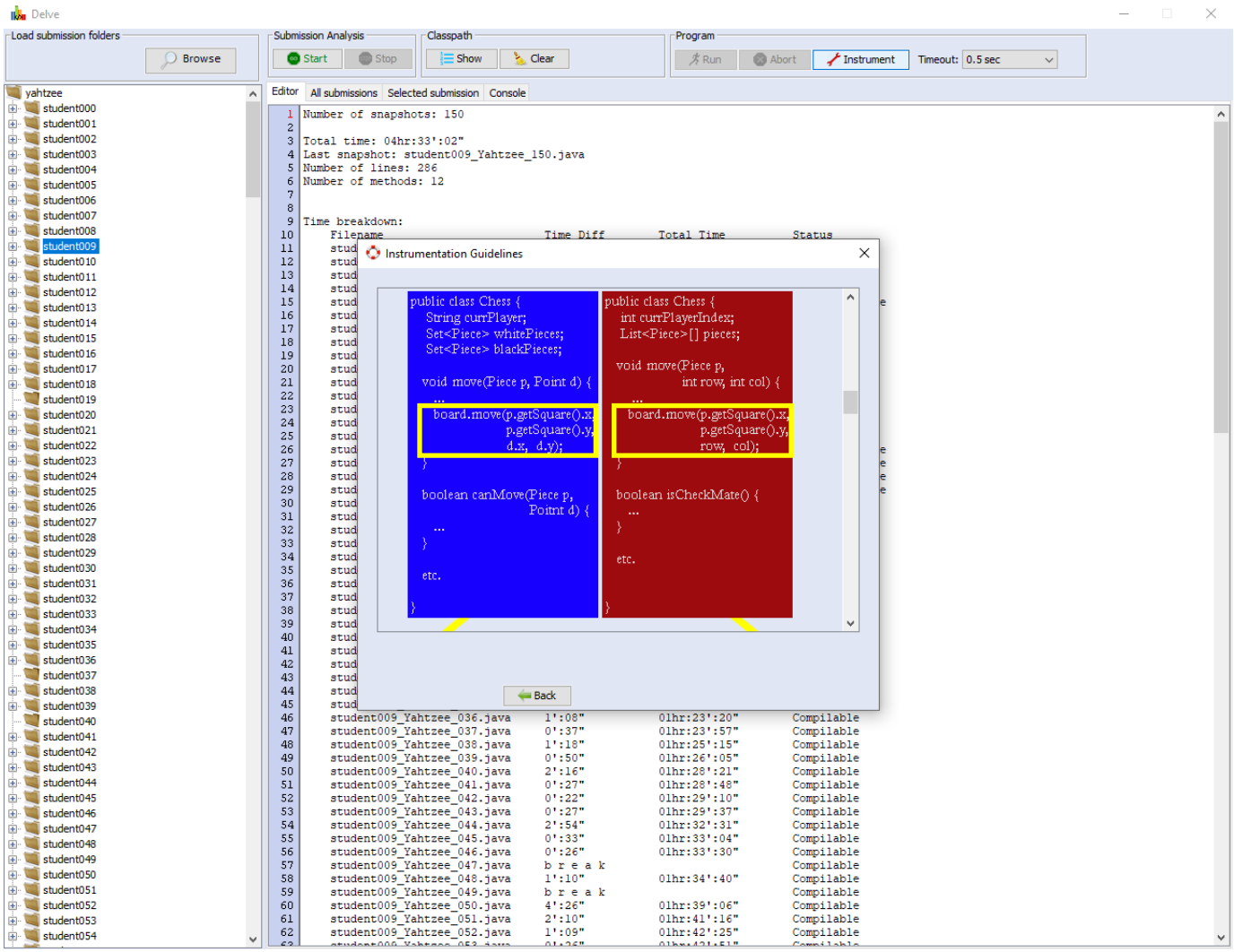

Figure 83: An example to show how to save the program state in cases when it is not so obvious.

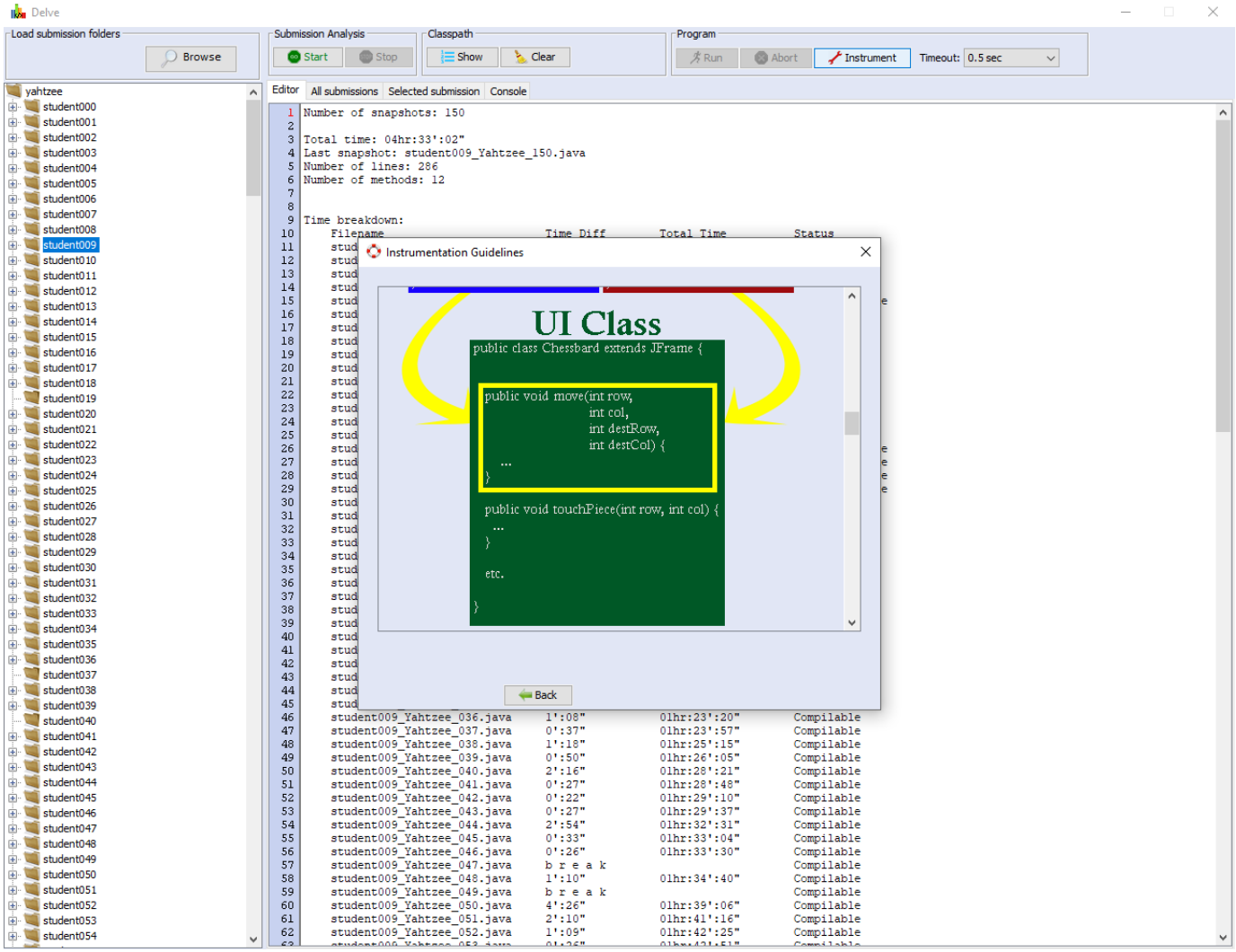

Figure 84: An example to show how to save the program state in cases when it is not so obvious.

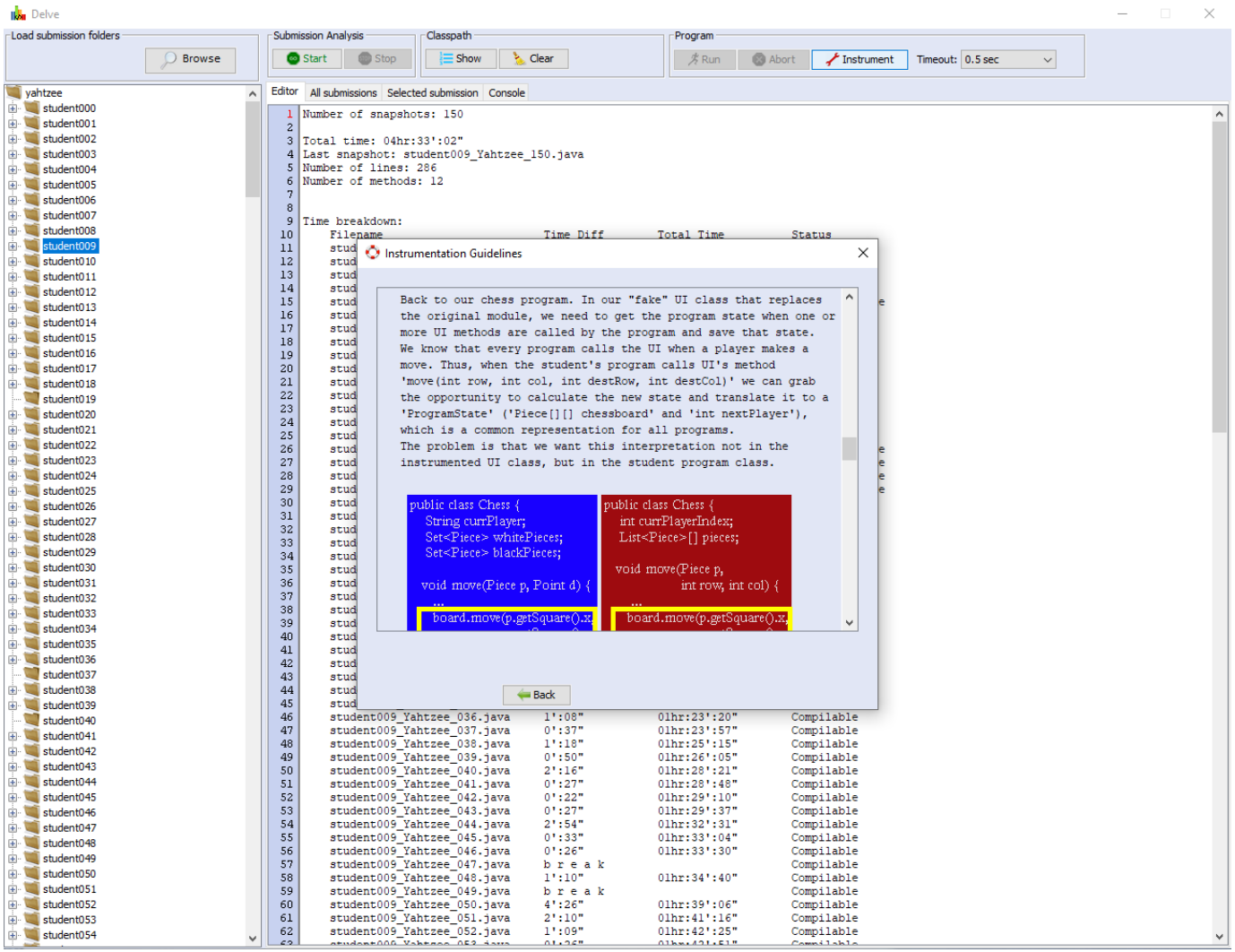

Figure 85: An example to show how to save the program state in cases when it is not so obvious.

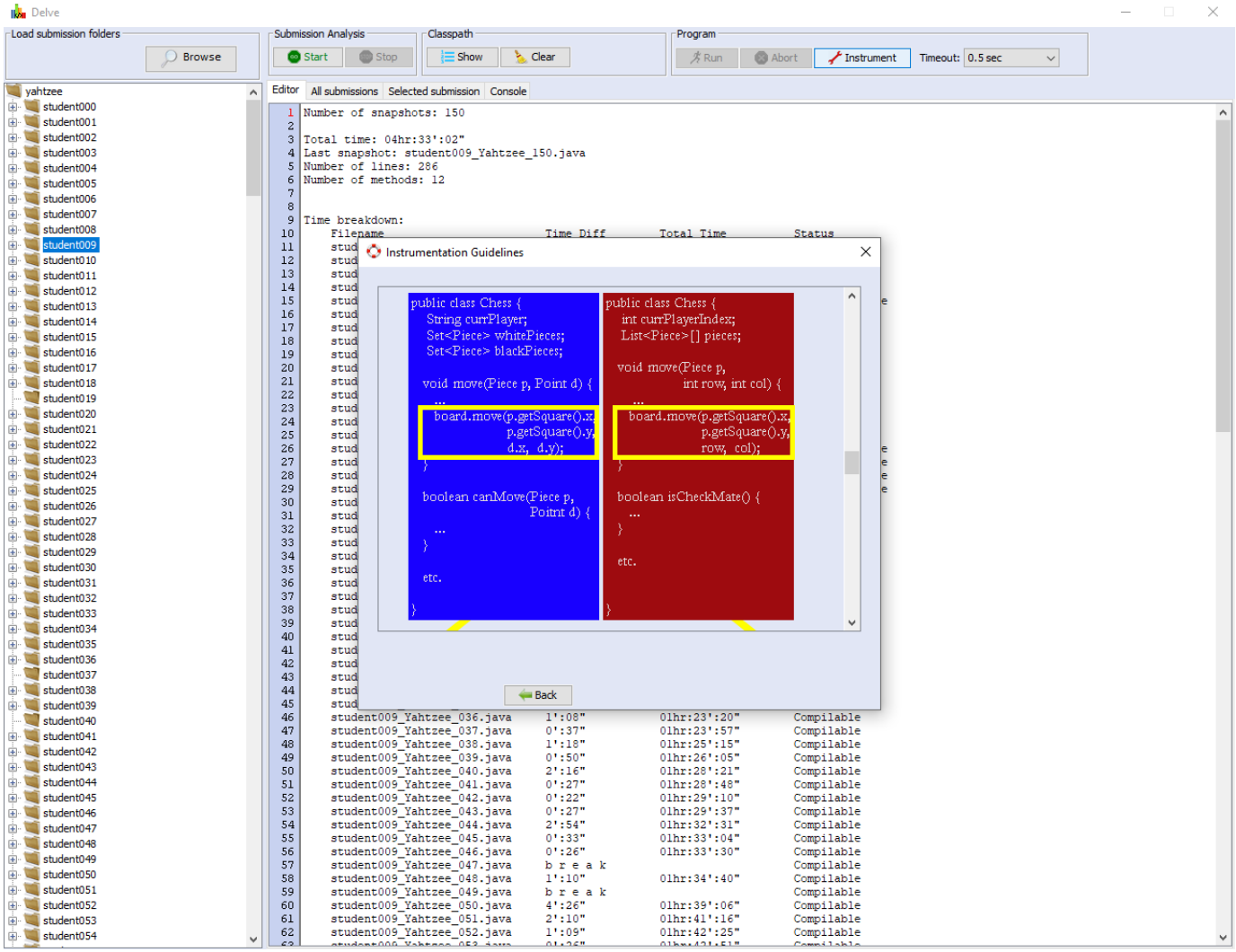

Figure 86: An example to show how to save the program state in cases when it is not so obvious.

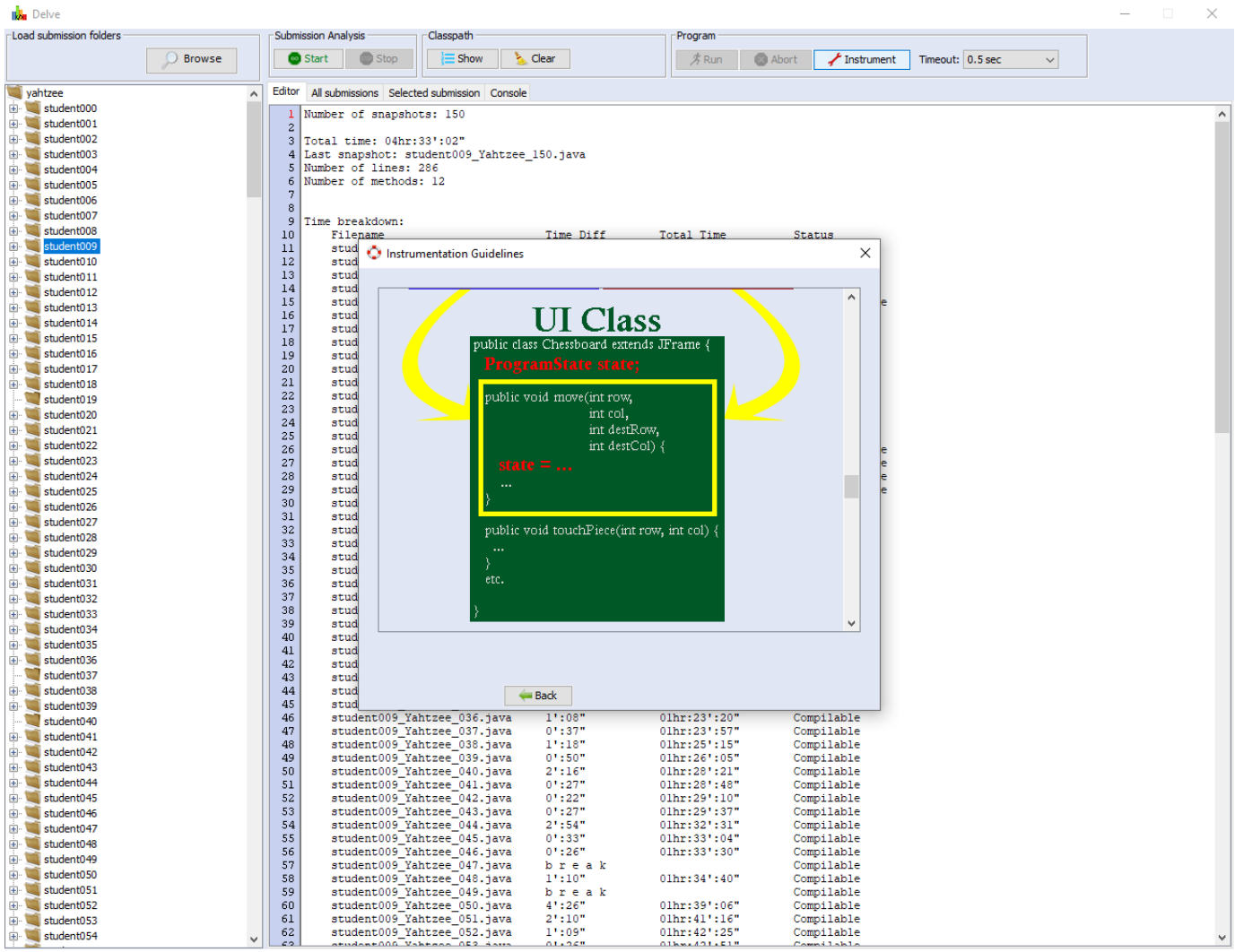

Figure 87: An example to show how to save the program state in cases when it is not so obvious.

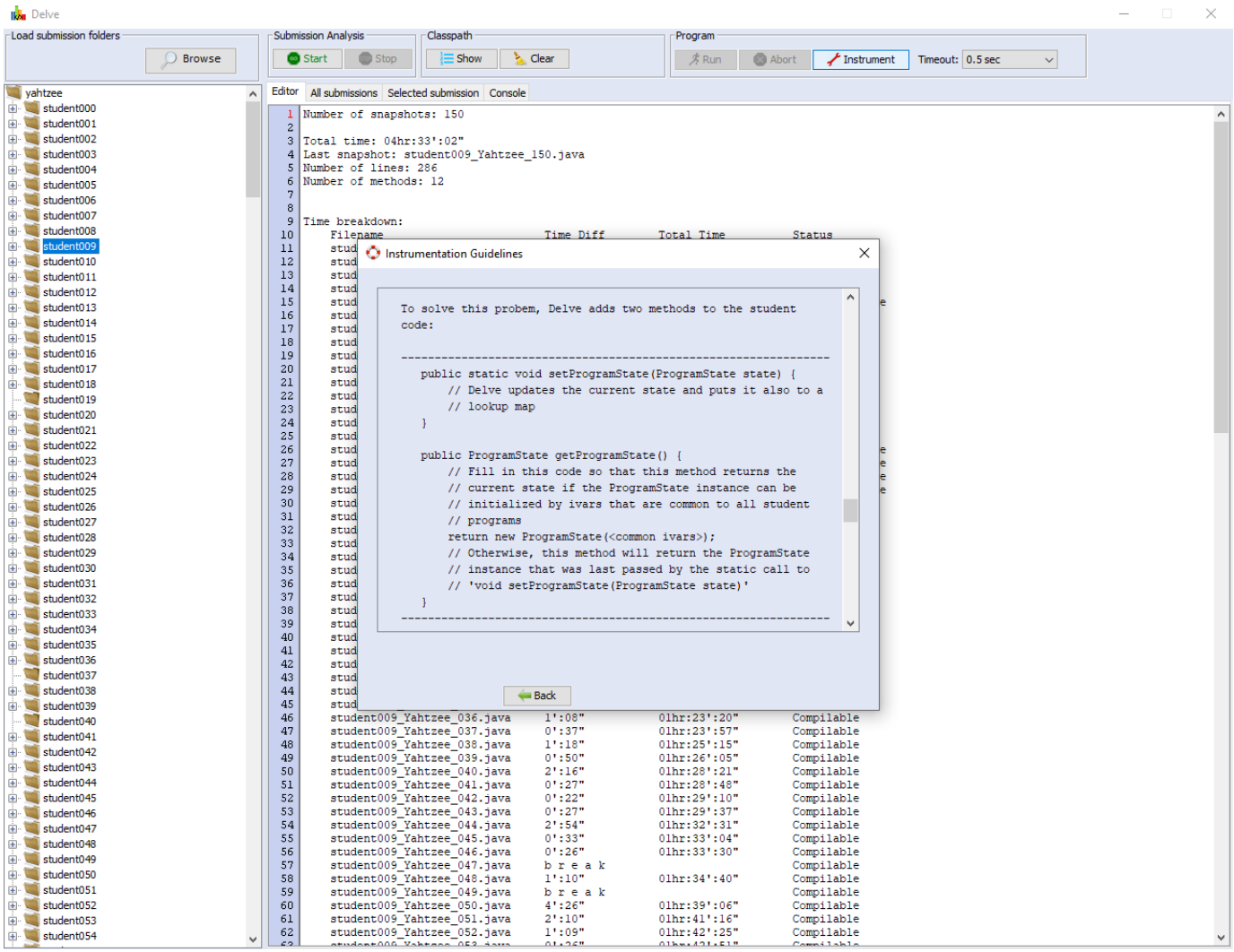

Figure 88: An example to show how to save the program state in cases when it is not so obvious.

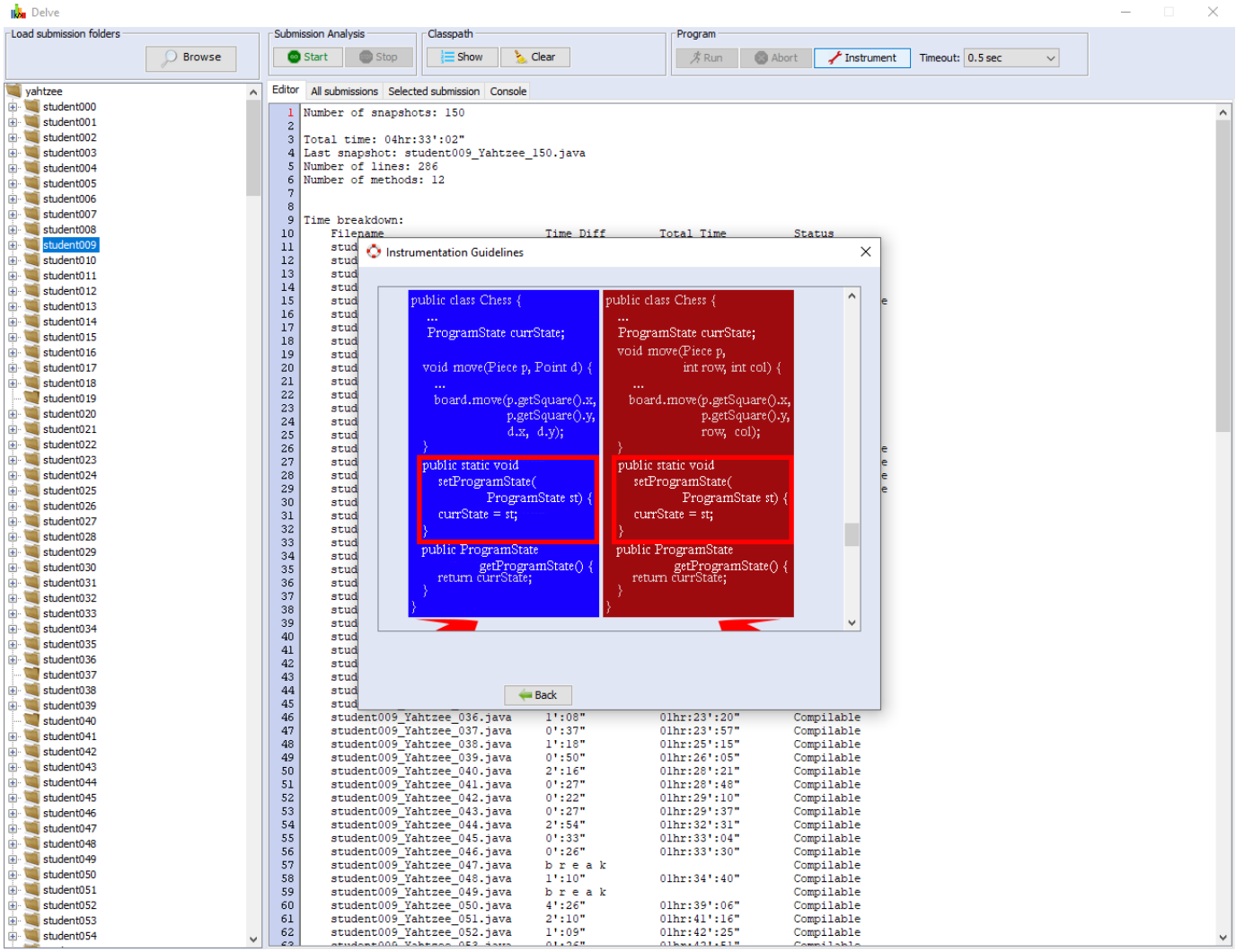

Figure 89: An example to show how to save the program state in cases when it is not so obvious.

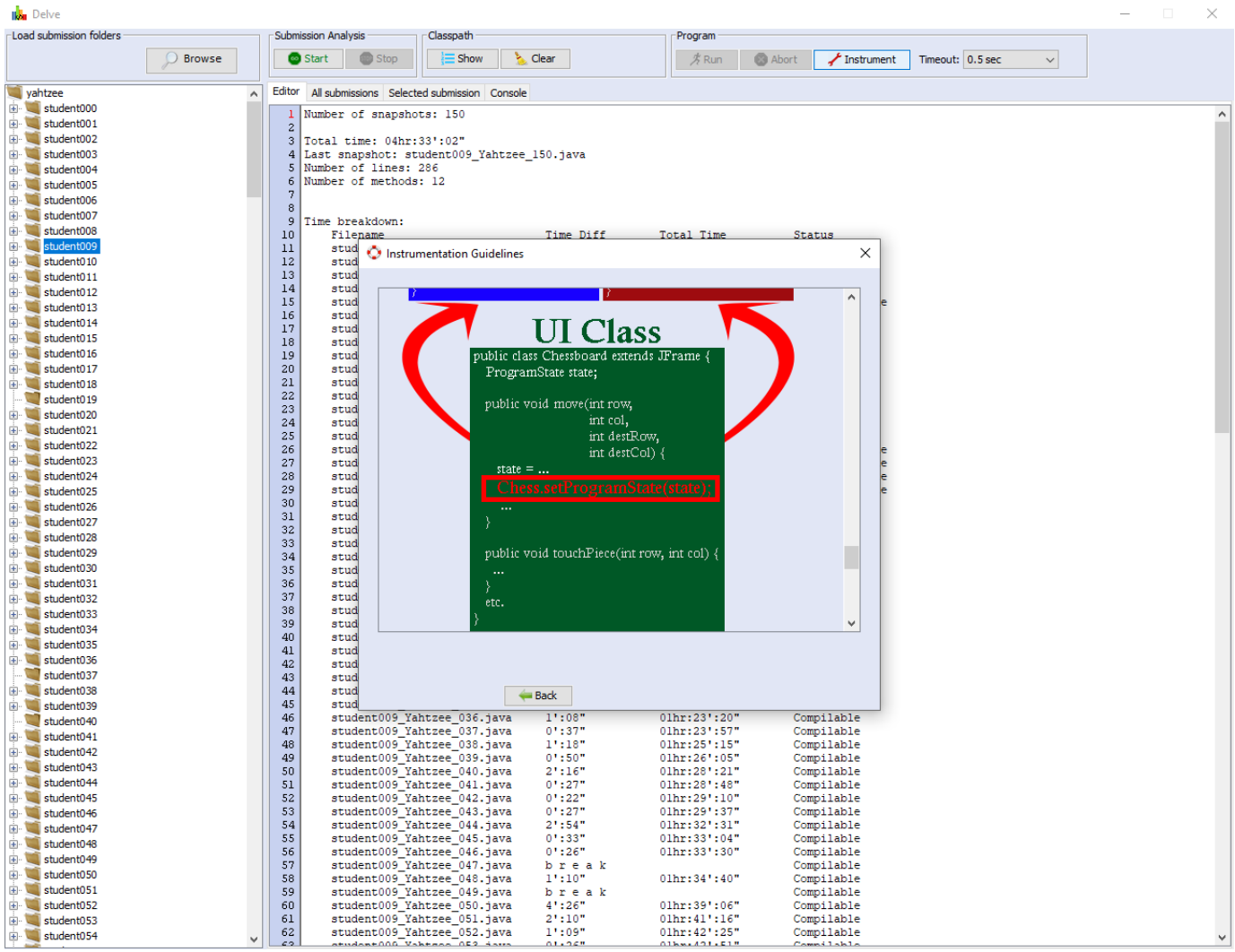

Figure 90: An example to show how to save the program state in cases when it is not so obvious.

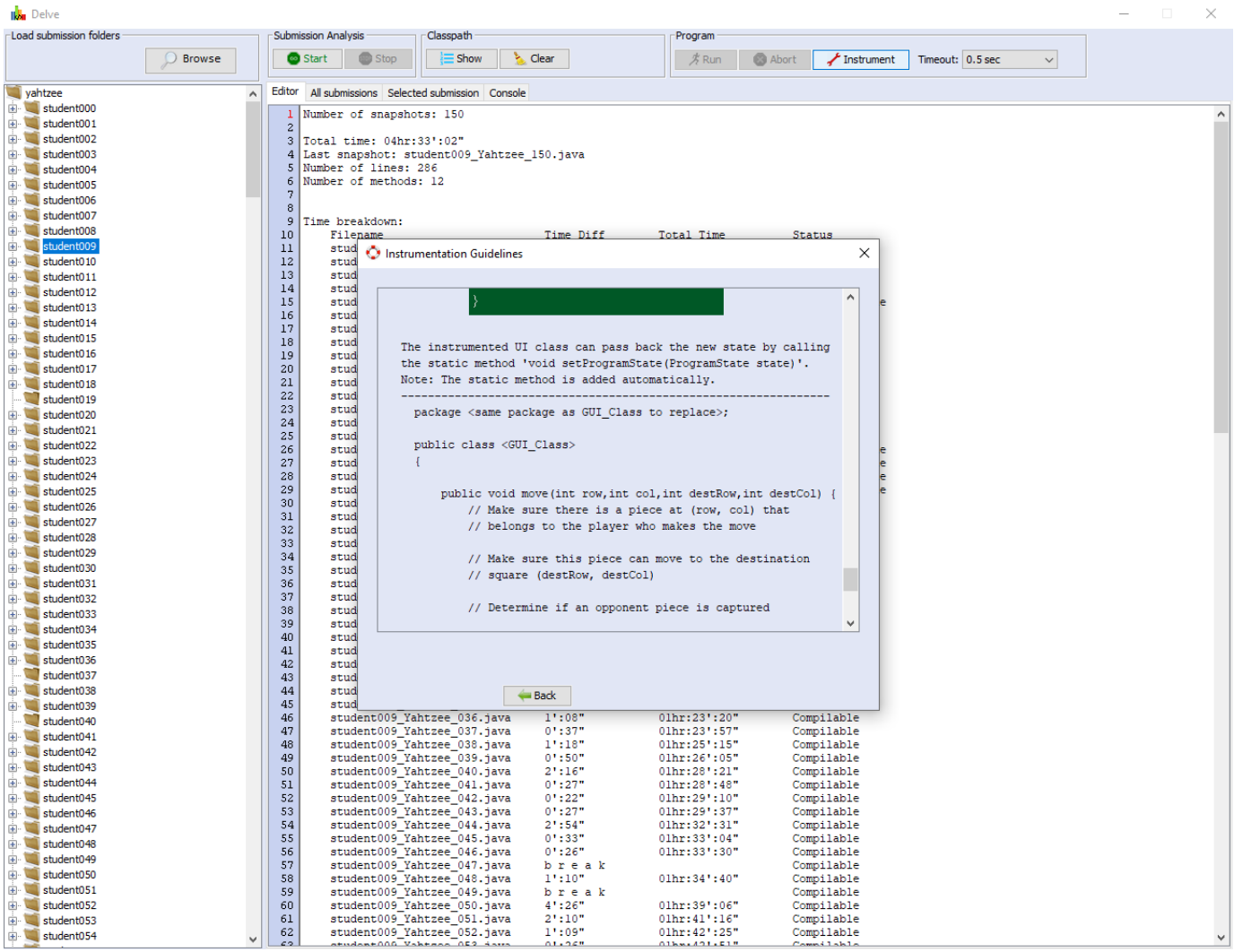

Figure 91: An example to show how to save the program state in cases when it is not so obvious.

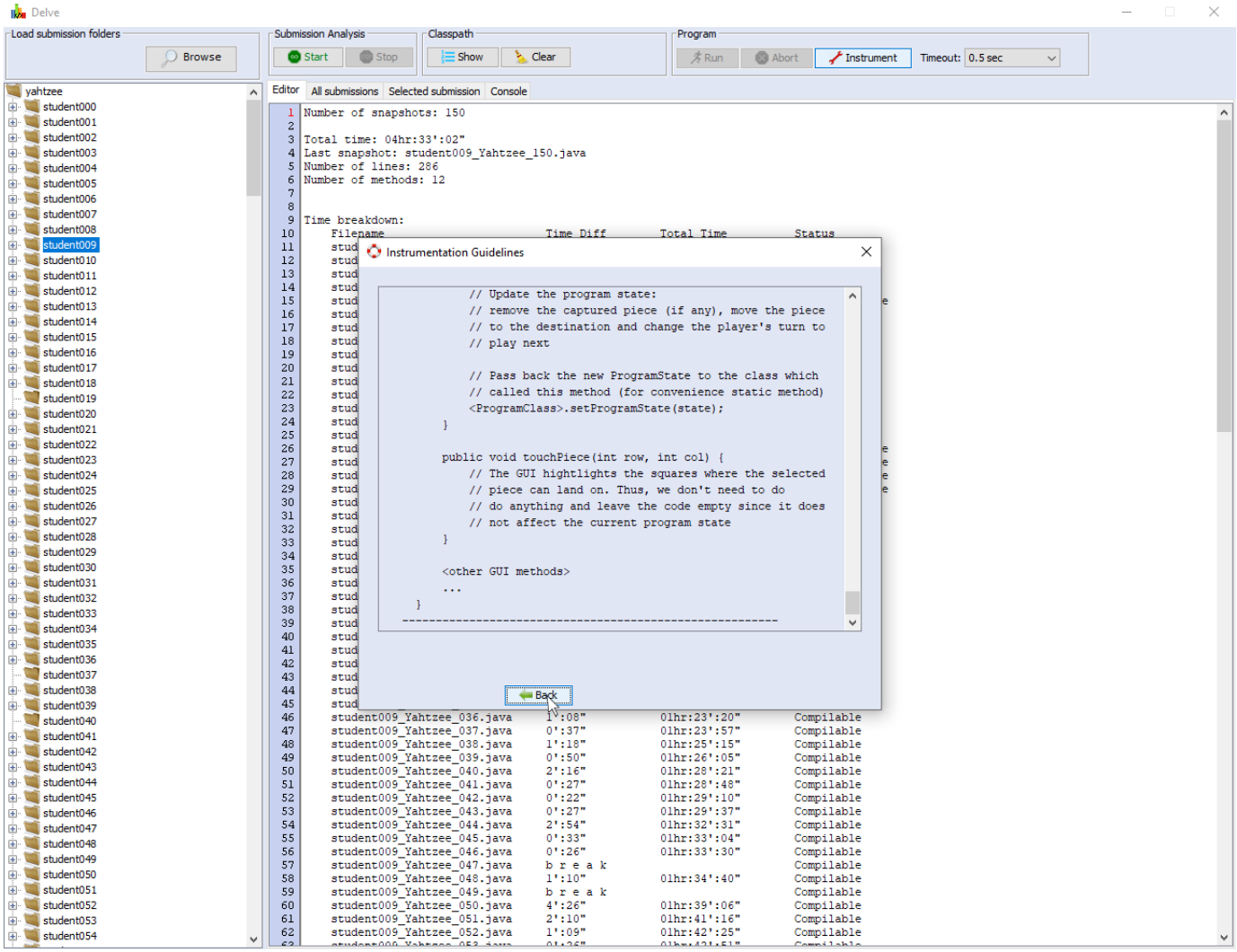

Figure 92: An example to show how to save the program state in cases when it is not so obvious.

In our example, we save the program state in one of the UI methods that are called in every program (see ***Figure 56***).

To add the code for the **`getProgramState()`** in the **`GetState`** class, the user can click the **`Add`** button.

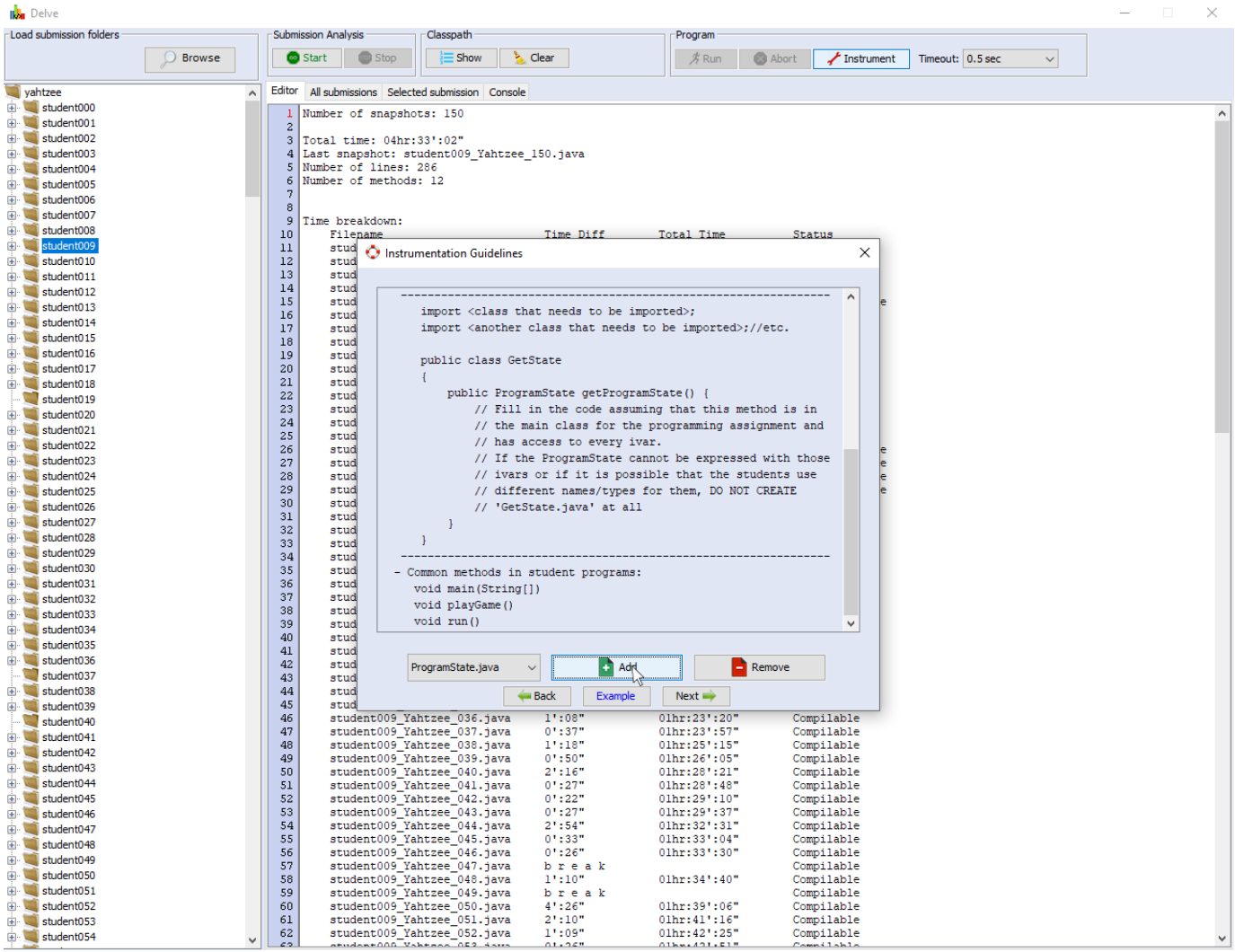

Figure 93: Click the *Add* button to locate file *GetState.java* which includes method *public ProgramState getProgramState()*.

and then locate file **`GetState.java`**.

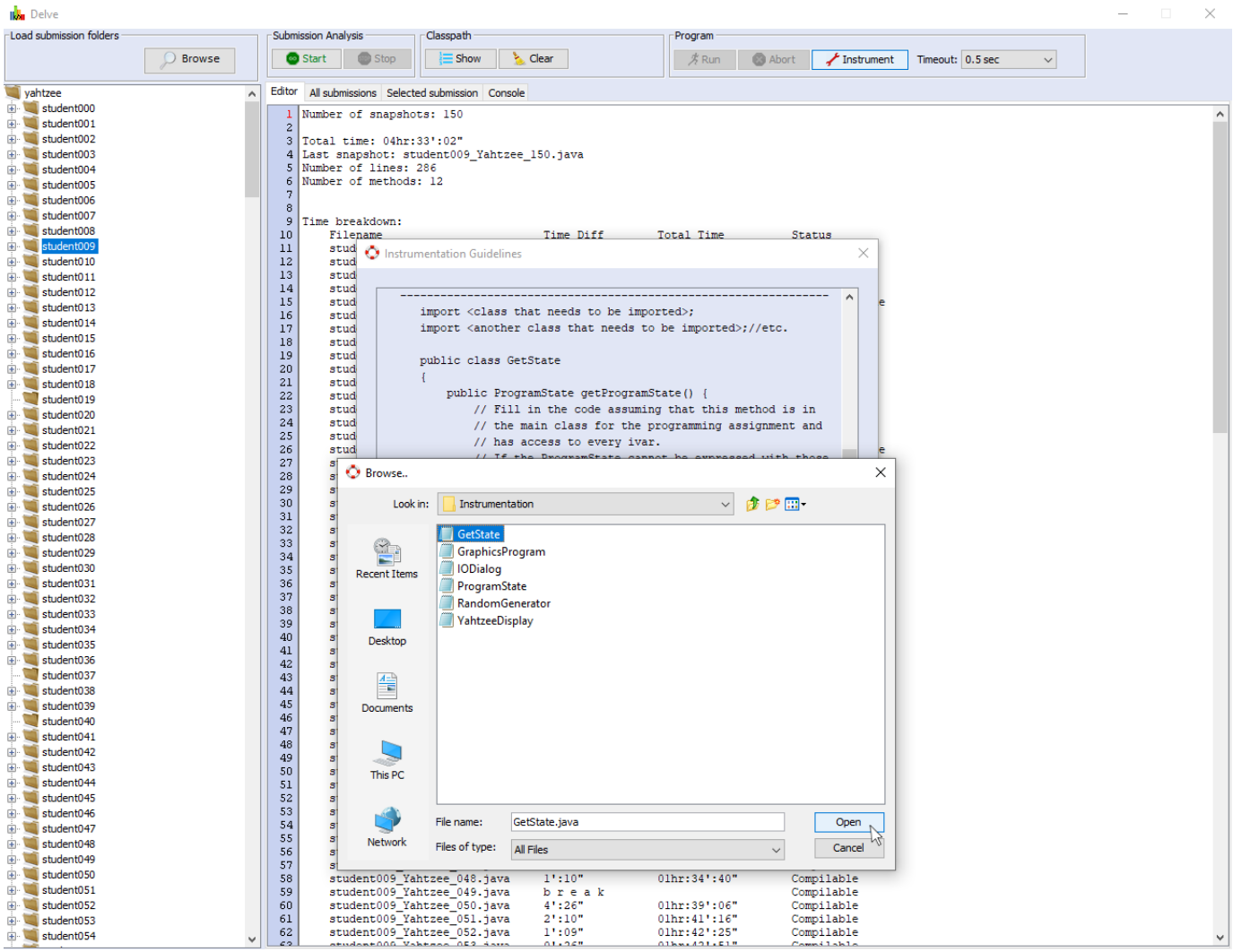

Figure 94: The user can now locate *GetState.java* to specify method *public ProgramState getProgramState()*.

In our example, the code for **`GetState.java`** is the default behavior which means that we could have omitted this step. This is useful for assignments where one can express the program state via common instance variables for all student submissions.

In most cases, **`GetState.java`** is unlikely to provide a different code to return the program state other than the default in ***Figure 95***.

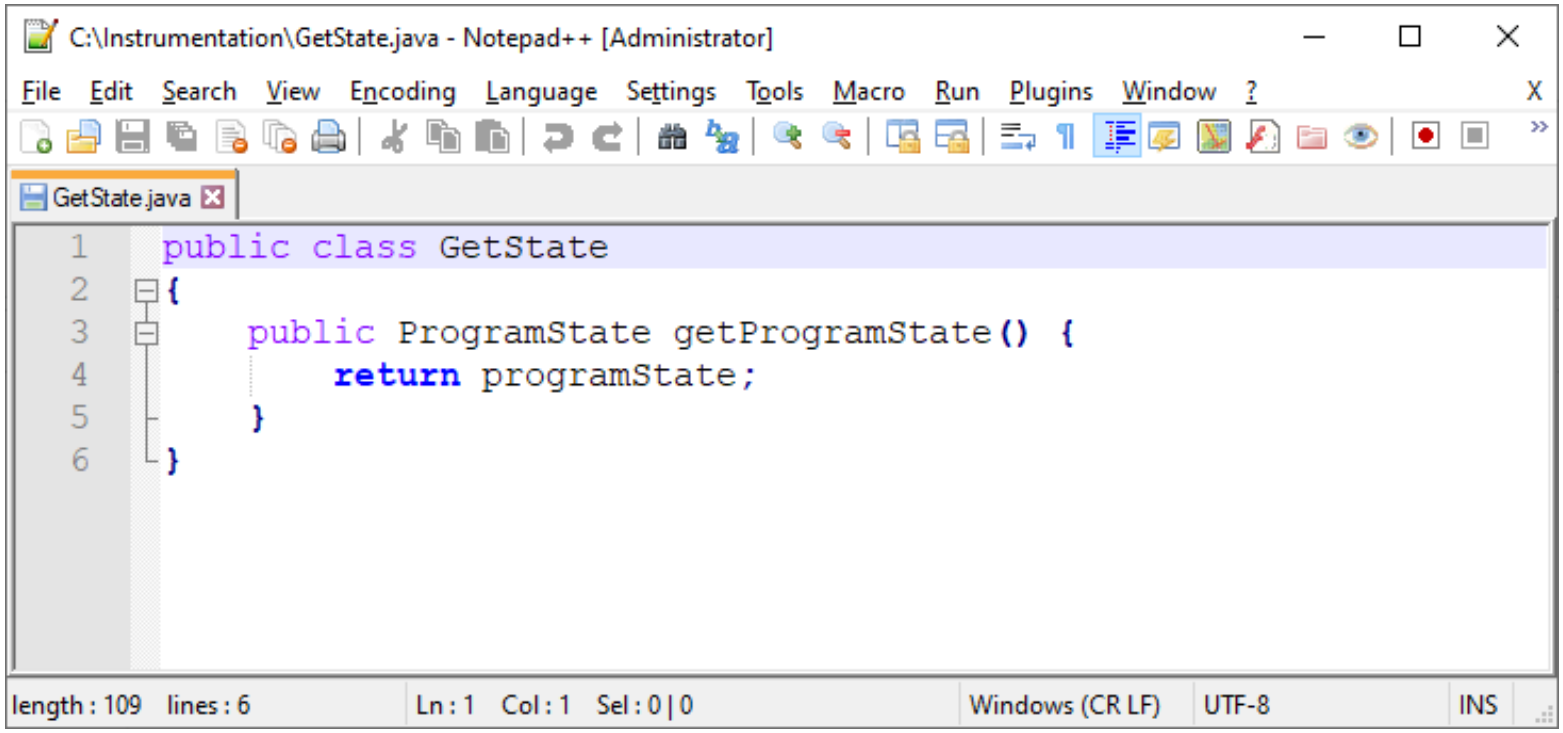

Figure 95: Default behavior for *GetState* (i.e., if *GetState.java* is not specified).

The goal is to set the program state in a method that all student programs call. This method can be in any class (e.g., UI class), which may not have direct access to the instance of the main module (in our example the main module is class Chess). To deal with that, Delve conveniently adds to the main module a public static method that we can access from any class: **`public static void setProgramState(ProgramState state)`** which is shown in ***Figure 89*** for our example. Then, the UI makes a direct call to it to set the program state (***Figure 90***).

At this point we have completed the fifth step and are ready to move to the next step.

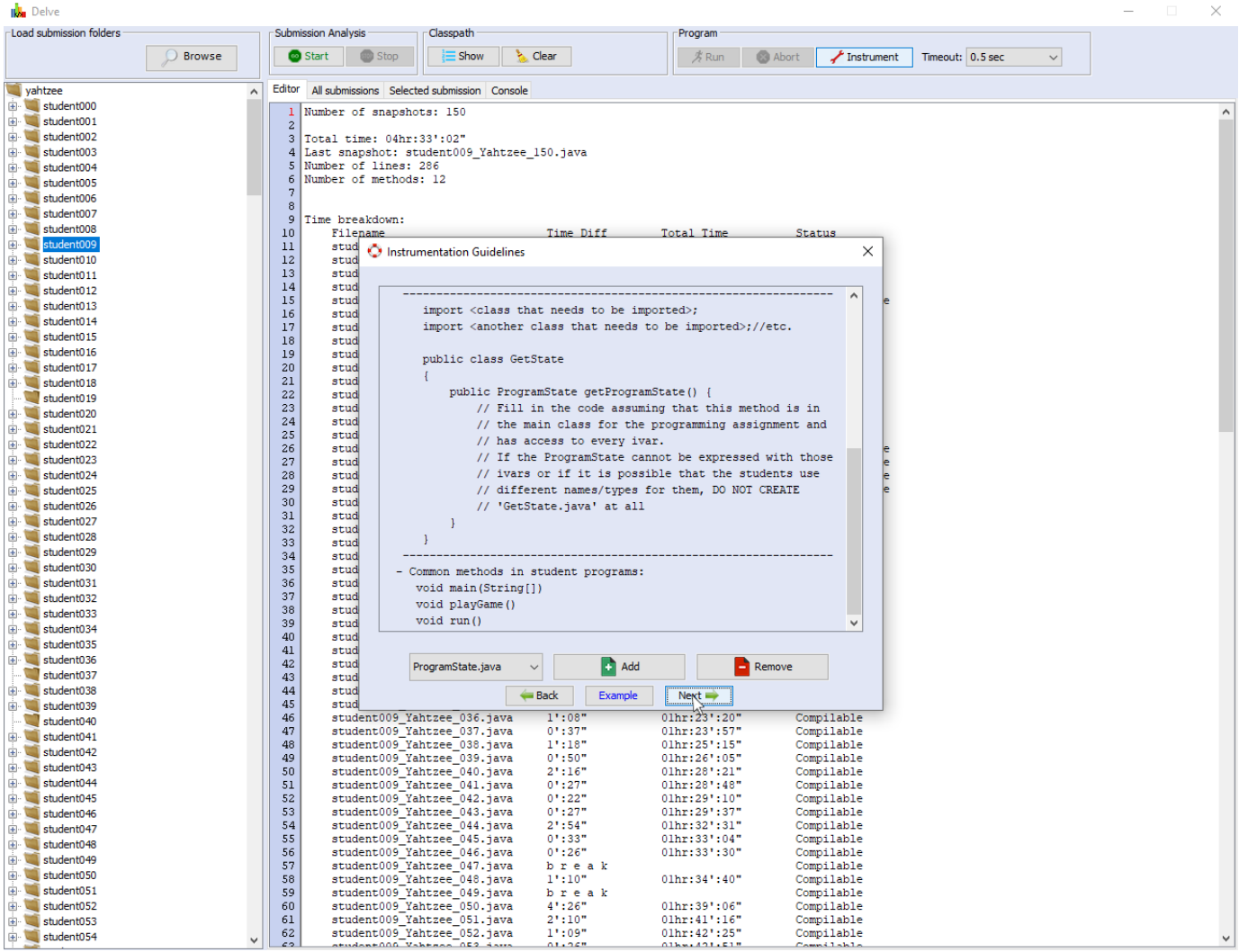

Figure 96: The fifth step is complete. Ready to move on to the sixth step.

7.7 Step 6: Provide a correct implementation as reference

Code instrumentation allows us to capture the states when the program executes. However, we do not know if a program state that gets produces is correct or not. To determine that we need a correct assignment implementation.

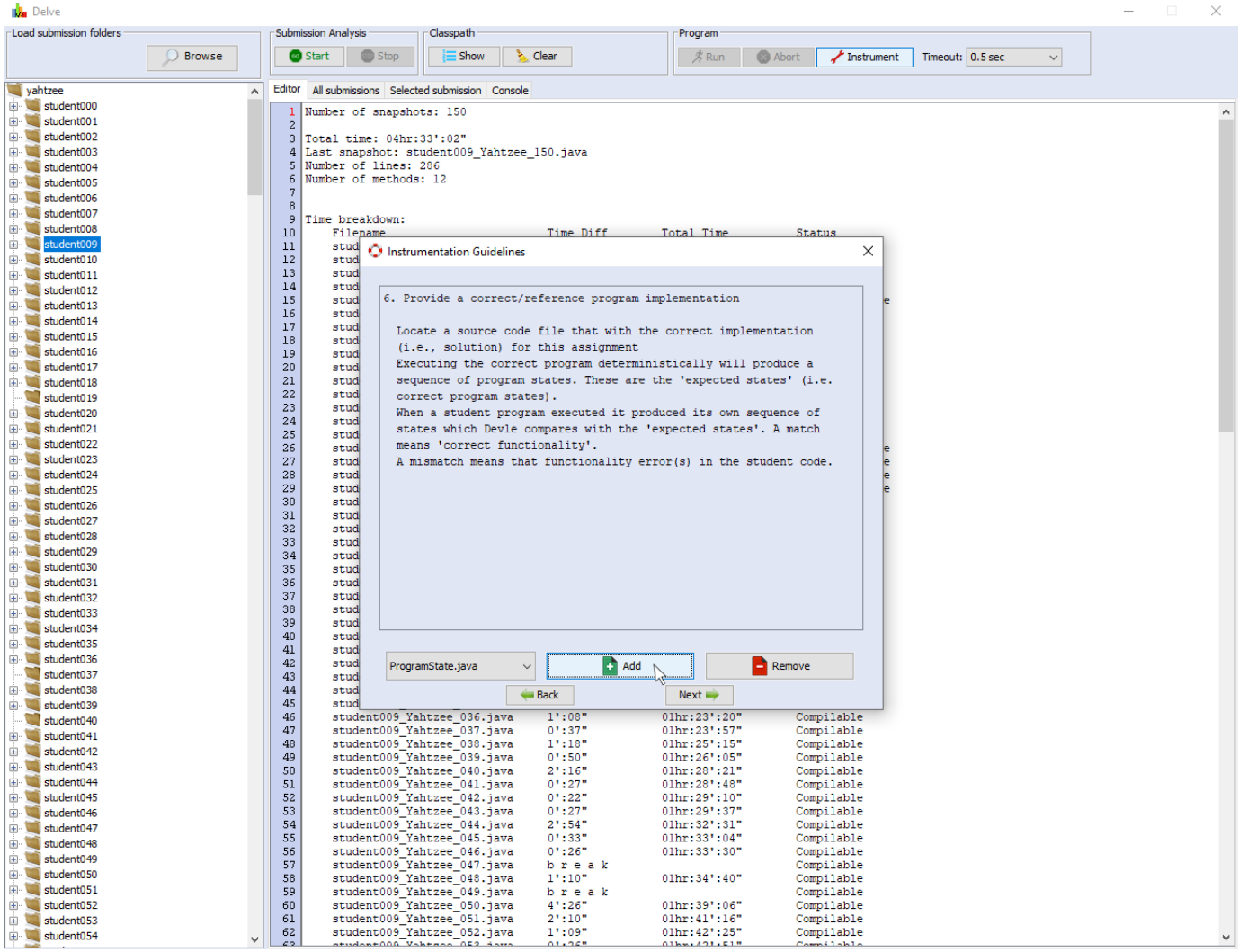

Figure 97: Click the *Add* button to locate the file with the correct implementation.

Delve instruments the code and executes it to produce a set of states. Deterministic nature means that every program runs with the same preconditions and therefore all programs that are correct should produce the same set of states.

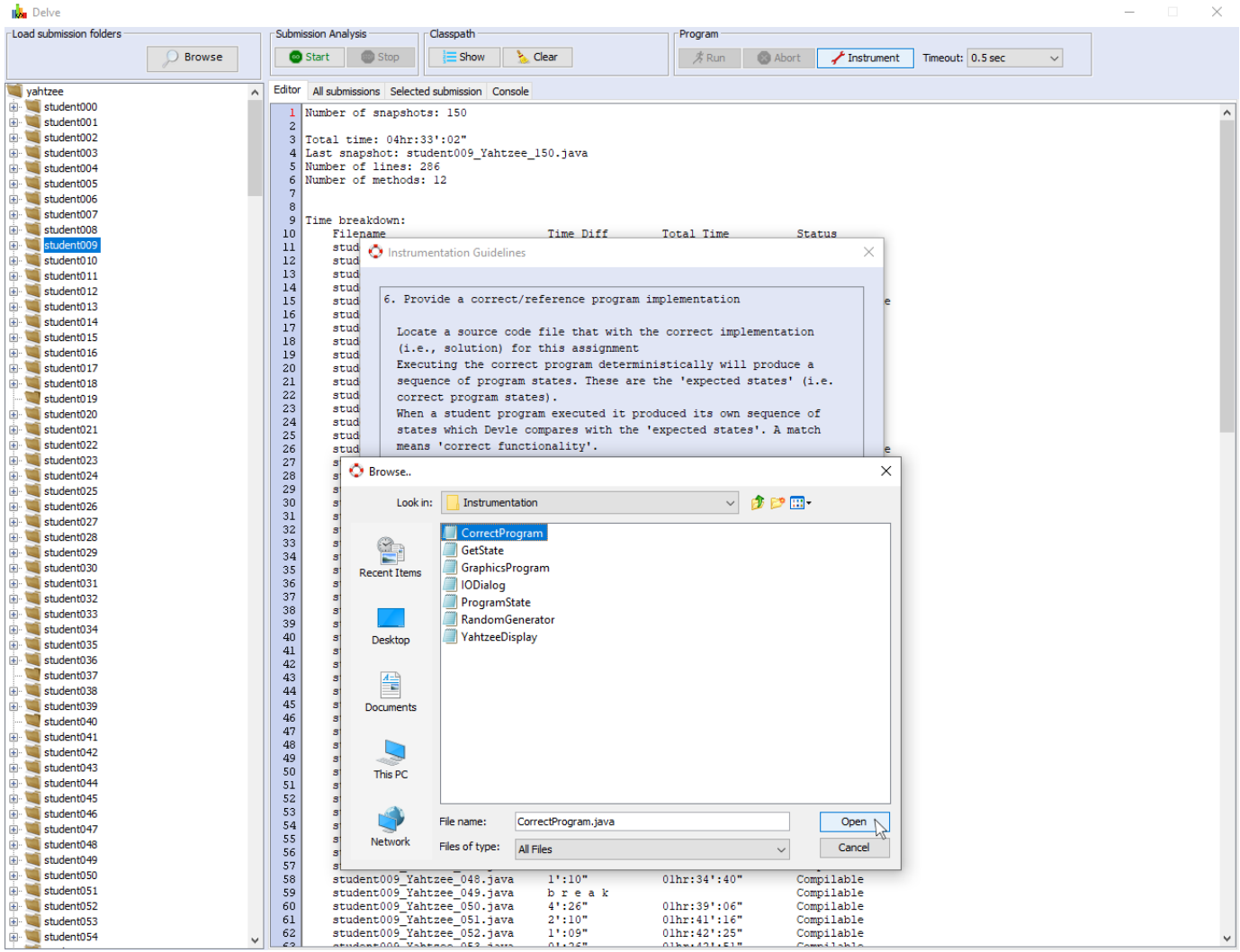

Figure 98: The user can now locate the program with the correct implementation.

At this point we have completed the sixth step and are ready to move to the next and final step.

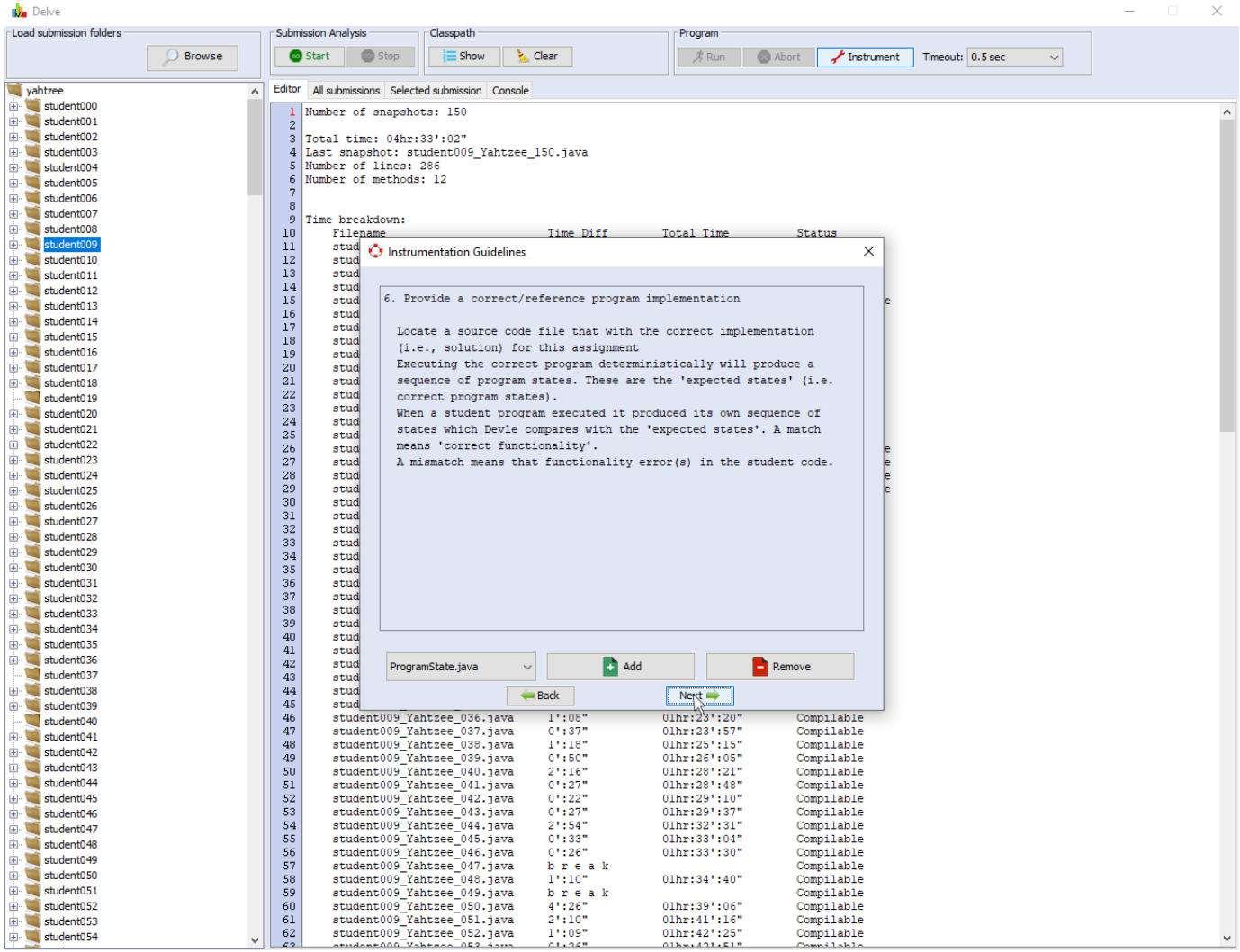

Figure 99: The sixth step is complete. Ready to move on to the last step.

7.8 Step 7: Provide additional libraries (if needed)

Rarely the extra code that is added for instrumentation uses a library (jar-file) that was not required before. If so, we must specify in this step the file(s) for those missing libraries.

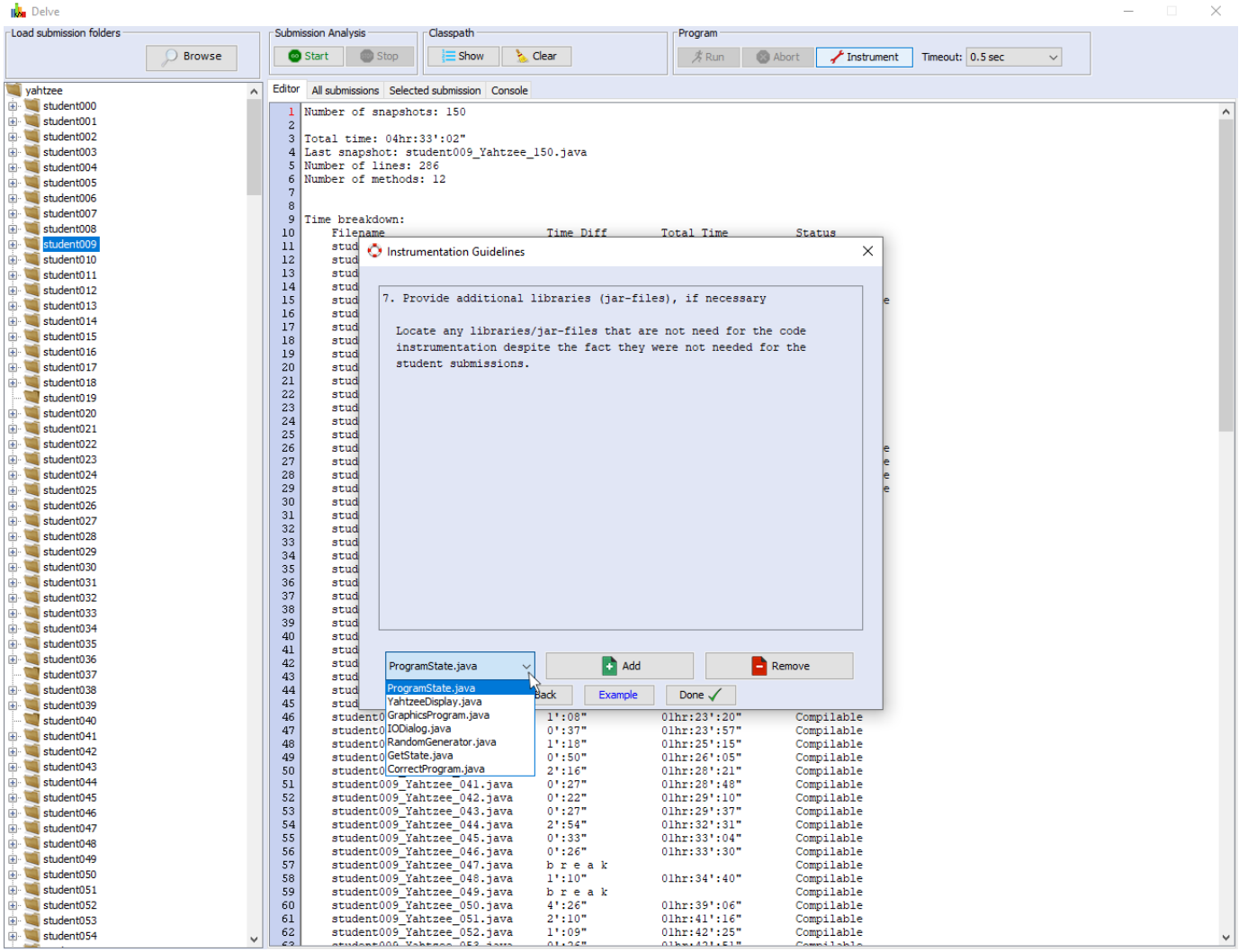

Figure 100: In our example there is no action in this step. The drop-down menu shows all files for the instrumentation.

At this point we are done with the instrumentation.

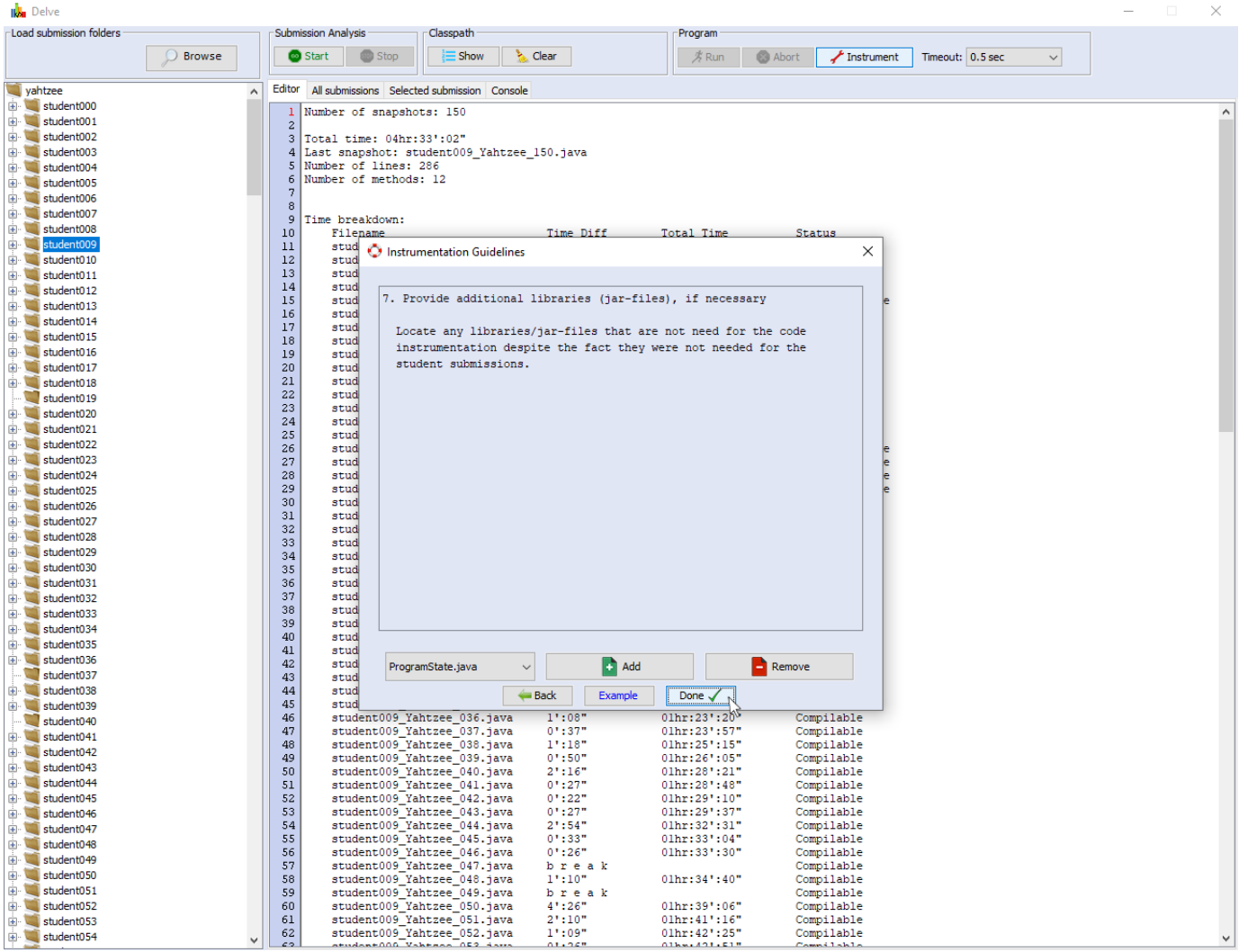

Figure 101: Click the *Done* button to begin the instrumentation analysis.

8. Instrumentation Analysis Results

Once done with the instrumentation setup and click Done (***Figure 101***), Delve starts the analysis which takes 2-3 minutes.

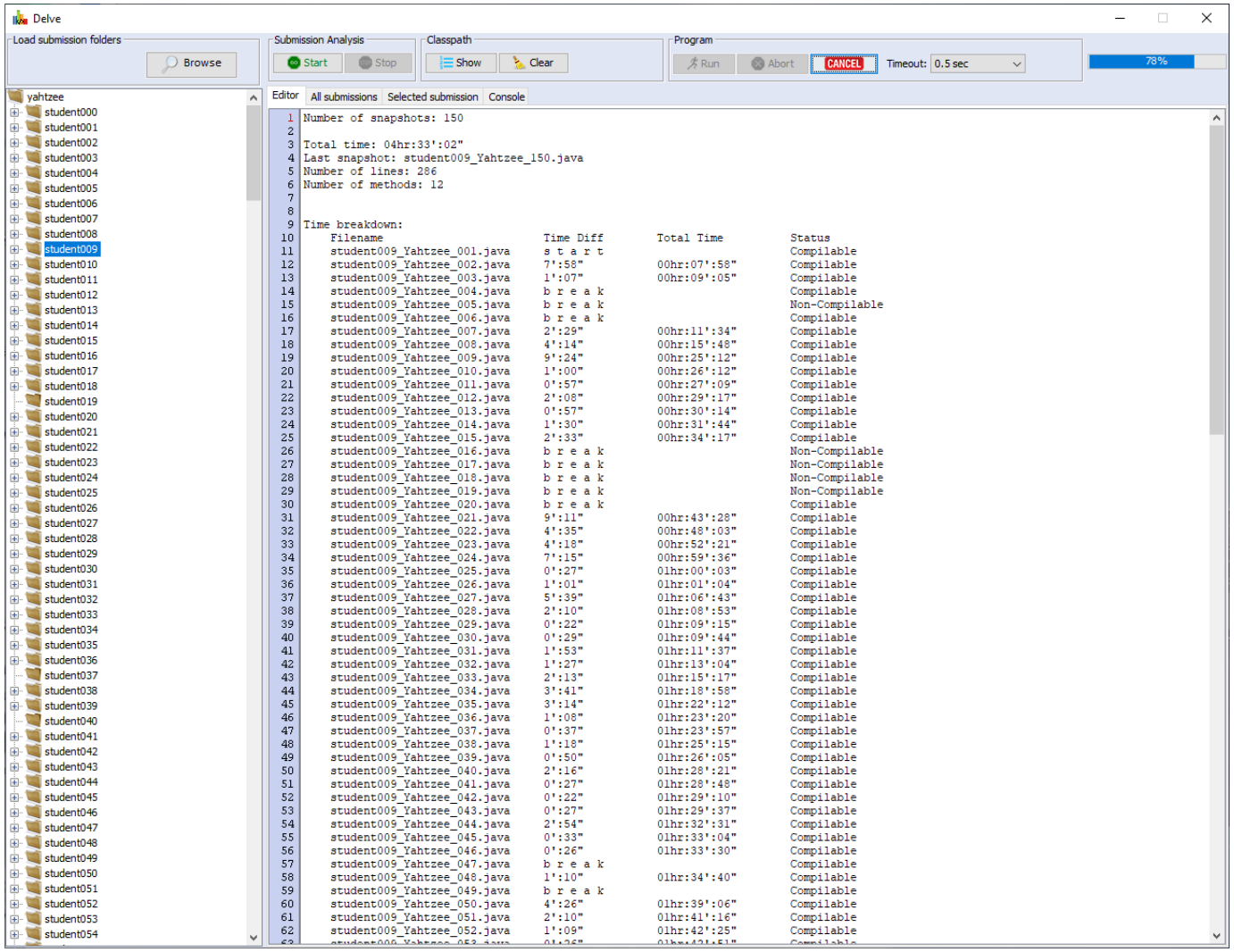

Figure 102: Once we are done with the instrumentation setup, Delve performs the analysis which takes a while.

Note that only programs that include **`public static void main(String[])`** can be executed. Delve, shows after the analysis an aggregated error message that lists the files that did not have this method.

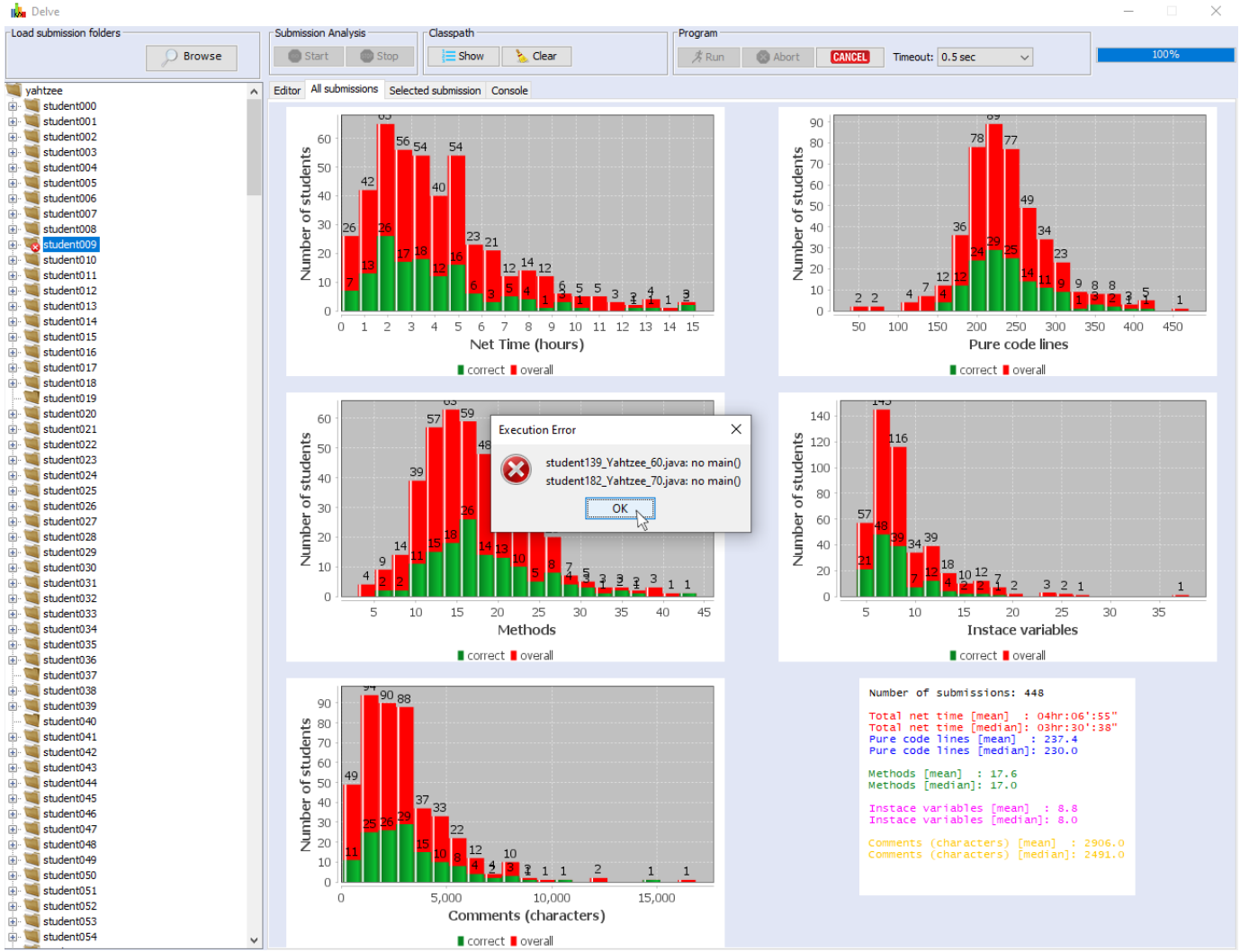

Figure 103: Only programs with *public static void main(String[])* can be analyzed. Delve lists the files that do not have main().

In the main panel, Delve plots the same histograms (in red) as before (***Figure 6***), but now it plots also (in green) the number of correct submissions for each bin. These plots provide a useful insight on things that work well and not so well.

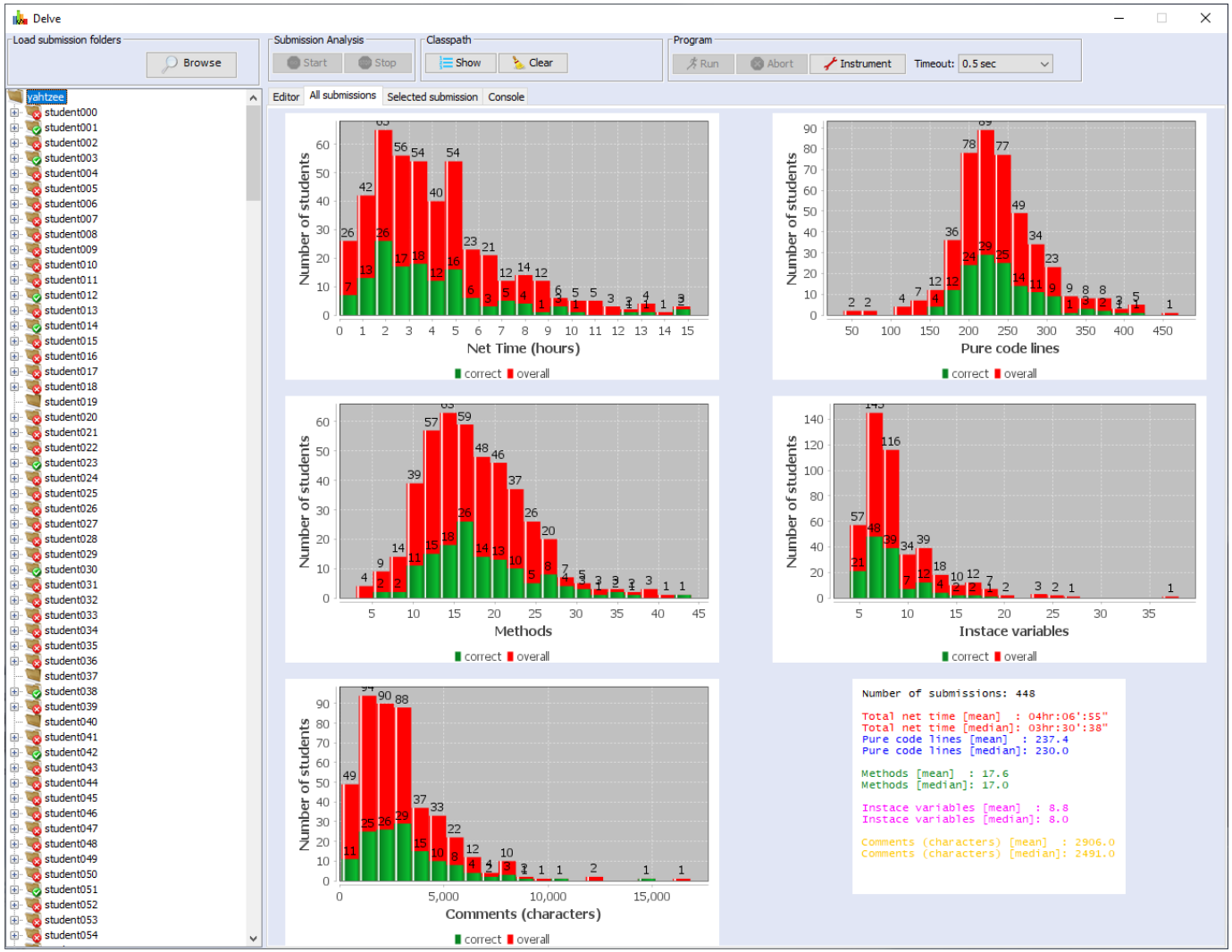

Figure 104: Histograms for correct submissions in contrast to the histrograms for all submissions provide useful insight.

At this point Delve, has graded every submission and the individual results are shown on the left, in the file tree. The red mark on the submission folder means that the submission was incorrect as opposed to the green mark. Note that there is no degree of correctness. Green means "*100% correct*" and red means "*not 100% correct*". The snapshot that is graded is also marked in green or red once we expand any given folder.

Figure 105: Delve's auto-grading notation.