Teaching and Learning Programming and Software Engineering via Interactive Gaming

Nikolai Tillmann
Microsoft Research
One Microsoft Way
Redmond, WA, USA
nikolait@microsoft.com

Jonathan de Halleux
Microsoft Research
One Microsoft Way
Redmond, WA, USA
jhalleux@microsoft.com

Tao Xie
Computer Science Dept.
NC State University
Raleigh, NC, USA
xie@csc.ncsu.edu

Sumit Gulwani
Microsoft Research
One Microsoft Way
Redmond, WA, USA
sumitg@microsoft.com

Judith Bishop
Microsoft Research
One Microsoft Way
Redmond, WA, USA
jbishop@microsoft.com

***Abstract*—Massive Open Online Courses (MOOCs) have recently gained high popularity among various universities and even in global societies. A critical factor for their success in teaching and learning effectiveness is assignment grading. Traditional ways of assignment grading are not scalable and do not give timely or interactive feedback to students. To address these issues, we present an interactive-gaming-based teaching and learning platform called Pex4Fun. Pex4Fun is a browser-based teaching and learning environment targeting teachers and students for introductory to advanced programming or software engineering courses. At the core of the platform is an automated grading engine based on symbolic execution. In Pex4Fun, teachers can create virtual classrooms, customize existing courses, and publish new learning material including learning games. Pex4Fun was released to the public in June 2010 and since then the number of attempts made by users to solve games has reached over one million. Our work on Pex4Fun illustrates that a sophisticated software engineering technique – automated test generation – can be successfully used to underpin automatic grading in an online programming system that can scale to hundreds of thousands of users.**

I. Introduction

Massive Open Online Courses (MOOCs) [8], [16] have recently gained high popularity among various universities and even in global societies. In these MOOCs, students come from a variety of geographical locations (typically globally) and may range in the thousands to hundreds of thousands. These MOOCs distribute the course materials such as recorded lectures, reading materials, and assignments via the Internet. Recently emerging companies such as Coursera [1], Udacity [5], and EdX [3] have been started to offer MOOCs in partnership with university professors individually or with universities officially.

For large courses at universities and especially for MOOCs, assignment grading is typically a critical factor for their success in teaching and learning effectiveness. Traditional ways of assignment grading by the teacher or teaching assistants are not scalable in such settings any more. In MOOCs, there are two main feasible ways of grading: automated grading and peer grading. Note that automated grading is typically feasible only for certain kinds of assignments such as programming-related exercises.

Automated grading relies on running a test suite (including a set of test cases) prepared by the teacher against each student solution. The grade for a student solution can be determined based on how high percentage of test cases from the test suite fail on the student solution. For example, the Coursera MOOC on "Software Engineering for SaaS" [2] taught by Fox and Patterson [8] adopted an auto-grader based on a test suite prepared by the teacher. However, the grading effectiveness heavily depends on the quality of the prepared test suite, which might not be of sufficient quality. A poor test suite could allow an incorrect student solution to pass all or most prepared test cases in the test suite and gain a full or high score. In reality, given the feedback information on the failing test cases, students could "over-fit" their solution to simply pass the failing test cases instead of striving for a correct solution.

Peer grading (by peer students) has been adopted as a grading mechanism at MOOCs (such as Coursera) for assignments such as essay writing. However, in general, the accuracy achieved by peer grading may not be satisfactory [9]. In the context of MOOCs, "resistance to peer grading", "concerns about privacy", "turning in blank assignments to get access to view the work of others", "the need for assessment to be part of a conversation" have been identified as issues with peer grading [14].

To address these issues in both MOOCs and large university courses, in this paper, we present an interactive-gaming-based teaching and learning platform called Pex4Fun (denoting Pex for Fun) for .NET programming languages such as C#, Visual Basic, and F#. Pex4Fun is publicly available at http://www.pexforfun.com/ and is a browser-based teaching and learning environment with target users such as teachers and students for introductory to advanced programming or software engineering topics. It works on any web-enabled device, even a smartphone. It comes with an auto-completing code editor, providing a user with instant feedback similar to the code editor in an integrated development environment such as Microsoft Visual Studio. Pex4Fun is a cloud application with the data in the cloud, enabling a user to use it anywhere where an Internet connection is available.

The key idea behind Pex4Fun is that there is a sample solution "under the hood" and the student learner is being encouraged to work towards this solution by iteratively supplying code. So close is this process to gaming, that Pex4Fun is

viewed by users as a game, with a byproduct of learning. Thus, new learners of programming can play games in Pex4Fun to master basic programming concepts. More advanced learners of software engineering can play games to master others skills such as skills of program understanding, induction, debugging, problem solving, testing, and specification writing. Teachers can create virtual classrooms in the form of courses by customizing existing learning materials and games or creating new materials and games. Teachers can also enjoy the benefits of automated grading of exercises assigned to students, making the platform applicable in the context of large courses at universities and MOOCs.

The core type of programming games is *coding duel* where the student has to solve a particular programming problem. Behind the scenes on the server in the cloud, the Pex4Fun website uses a technique called dynamic symbolic execution [10], [18] implemented by the automated test-generation tool called Pex [24], in order to determine the progress of the student and to compute customized feedback. In a coding duel, the student player is given a player implementation, being an empty or faulty implementation of a method (sample solution), with optional comments to give the player hints in order to reduce the difficulty level of gaming. (Note that in a regular classroom setting such as in a MOOC or university course, the behavioral requirements for revising the player implementation would be specified in the comments.) Then the player is asked to modify the player implementation to make its behavior (in terms of the method inputs and results) to be the same as the secret sample-solution implementation, which is supplied by the game creator but is not visible to the player. During the game-playing process, the player has the opportunity to request the gaming platform to provide feedback as to the method input(s) that cause the player implementation and the secret implementation to have the same or different method results. The gaming platform leverages Pex to provide such feedback. It is in the nature of Pex to provide a small number of inputs/results that are representative for the correct and incorrect aspects of the student's player implementation. As a result, the student is not overwhelmed by details and yet receives relevant feedback to proceed.

The game type of coding duels within Pex4Fun is flexible enough to allow game creators to create various games to target a range of skills such as programming, program understanding, induction, debugging, problem solving, testing, and specification writing, with different difficulty levels of gaming. In addition, Pex4Fun is an open platform: any one around the world can create coding duels for others to play besides playing existing coding duels themselves. The platform also provides various features to engage students in a social learning environment such as ranking of players and coding duels (http://www.pexforfun.com/Community.aspx) and online live feeds (http://www.pexforfun.com/Livefeed.aspx).

Pex4Fun was adopted as a major platform for assignments in a graduate software engineering course. A coding-duel contest was held at a major software engineering conference (ICSE 2011) for engaging conference attendees to solve coding duels in a dynamic social contest. Pex4Fun has been gaining high popularity in the community: since it was released to the public in June 2010, the number of clicks of the "Ask Pex!" button (indicating the attempts made by users to solve games at Pex4Fun) has reached over one million (1,135,931) as of March 3, 2013. Pex4Fun has provided a number of open virtual courses (similar to MOOCs in spirit) including learning materials along with games used to reinforce students' learning (http://www.pexforfun.com/Page.aspx#learn/courses).

Our work on Pex4Fun illustrates that a sophisticated software engineering technique – automated test generation – can be successfully used to underpin automatic grading in an online programming system that can scale to hundreds of thousands of users. It makes a real contribution to the known problem of assignment grading, as well as providing a programming-oriented gaming experience outside of the classroom.

The rest of the paper is organized as follows. Section II presents related work. Section III presents the background information. Section IV presents the concept of coding duels, the major type of games within Pex4Fun. Section V discusses the design goals and principles for coding duels. Section VI presents Pex4Fun's teaching support. Section VII presents example coding-duel exercises in a course. Sections VIII and IX discuss initial experiences on adopting Pex4Fun in a course and an open contest of coding duels, respectively. Section X concludes with future work.

II. Related Work

Jackson and Usher [12] developed ASSYST to assist a tutor to assess student solutions along with student tests for their solutions (written in the Ada programming language). ASSYST relies on test cases provided by the tutor to assess the correctness and efficiency of the student solutions. It also provides static analysis to assess the efficiency, style, and complexity of the student solutions, and provides code-coverage measurement to assess the adequacy of the student tests. Zeller [28] developed Praktomat to assess student solutions based on test cases provided by the teacher. Praktomat also provides features for allowing students to read, review, and assess each other's solutions in the style of peer grading. Edwards and Perez-Quinones [7] developed Web-CAT (http://web-cat.sourceforge.net/) to assess student solutions and student tests for their solutions (written in Java). Web-CAT also relies on test cases provided by the teacher to assess student solutions. Web-CAT shares similar design goals of Praktomat or ASSYST and further provides advanced features to support test-driven development. Unlike ASSYST or Web-CAT, Pex4Fun currently does not focus on assessing student tests but focuses on assessing student solutions. There, tests generated by Pex (underlying Pex4Fun) are not a fixed set, unlike the tests used by ASSYST, Praktomat, or Web-CAT: Pex generates different tests depending on the program behavior of a student's submitted player implementation, accomplishing the goal of personalized or customized feedback for each student.

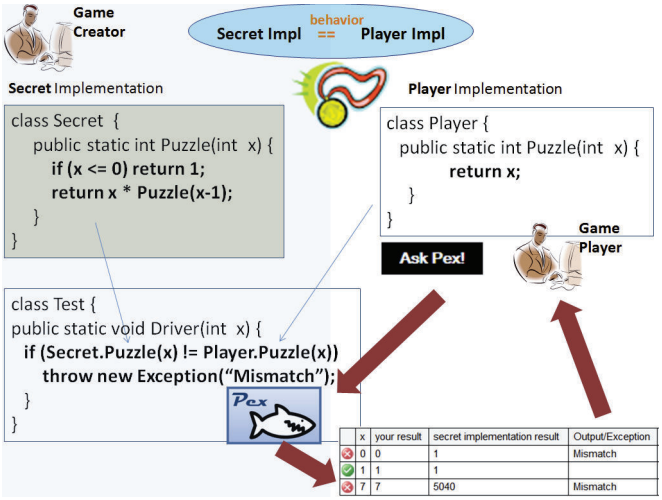

Fig. 1. The workflow of creating and playing a coding duel

Spacco et al. [21] developed Marmoset (http://marmoset.cs.umd.edu/), an automated snapshot, submission, and testing system. The system captures snapshots of students' programming projects to a centralized code repository whenever the students save their source files. Such a collected fine-grained revision history offers teachers a unique perspective into the development process for students. Similar to this aspect, our proposed platform also captures snapshots of students' revisions of the player implementation whenever the students click the "Ask Pex!" button to request feedback from Pex. Similarly, teachers can investigate the duel-solving processes by students. When using Marmoset, students can also explicitly submit projects to the Marmoset system to request Marmoset to run these submissions against a suite of unit tests prepared by the teachers to evaluate the functional correctness of a submission. In contrast, when using Pex4Fun, the teachers are not required to prepare a suite of tests for evaluating functional correctness of a player implementation against the secret implementation. Instead, Pex is used to serve this evaluation purpose. In addition, tests generated by Pex are not a fixed set, unlike the tests used by Marmoset.

III. Background

We next present the underlying technology (dynamic symbolic execution) and supporting tool (Pex) for the Pex4Fun platform. Dynamic symbolic execution (DSE) [10], [18] is a variation of symbolic execution [6], [15] and leverages runtime information from concrete executions. DSE is often conducted in iterations to systematically increase code coverage such as block or branch coverage. In each iteration, DSE executes the program under test with a test input, which can be a default or randomly generated input in the first iteration or an input generated in one of the previous iterations. During the execution of the program under test, DSE performs symbolic execution in parallel to collecting symbolic constraints on program inputs obtained from predicates in branch statements along the execution. The conjunction of all symbolic constraints along an executed path is called the path condition. Then DSE flips a branching node in the executed path to construct a new path that shares the prefix to the node with the executed path, but then deviates and takes a different branch. DSE relies on a constraint solver to (1) check whether such a flipped path is feasible; if so, (2) compute a satisfying assignment — such assignment forms a new test input whose execution will follow the flipped path.

Based on dynamic symbolic execution, Pex [24] is an automatic white-box test-generation tool for .NET, which has been integrated into Microsoft Visual Studio as an add-in. Besides being adopted in industry, Pex has been used in classroom teaching at different universities, as well as various tutorials both within Microsoft (such as internal training of Microsoft developers) and outside Microsoft (such as tutorials at .NET user groups) [27].

A key methodology that Pex supports is parameterized unit testing [25]–[27], which extends the current industry practices based on closed, traditional unit tests (i.e., unit test methods without input parameters). In parameterized unit testing, unit test methods are generalized by allowing parameters to form parameterized unit tests. This generalization serves two main purposes. First, parameterized unit tests are specifications of the behavior of the methods under test. They include exemplary arguments to the methods under test, and the ranges of such arguments. Second, parameterized unit tests describe a set of traditional unit tests that can be obtained by instantiating the methods of the parameterized unit tests with given argument-value sets. An automatic test-generation tool such as Pex can be used to generate argument-value sets for parameterized unit tests.

IV. Coding Duels

Coding duels are the major type of games within the Pex4Fun platform for learning various concepts and skills in programming or software engineering. Figure 1 shows the workflow of creating and playing an example coding duel. Figure 2 shows a screen snapshot of the user interface of the Pex4Fun website, which shows the example coding duel being solved by a player.

In a coding duel, a player's task is to implement the `Puzzle` method (shown on the top-right side of Figure 1 and in Figure 2) to have exactly the same behavior as another secret `Puzzle` method, which is never shown to the player (shown on the top-left side of Figure 1), based on feedback in the form of some selected values where the player's current version of the `Puzzle` method behaves differently as well as some selected values where it behaves the same way (shown near the right-bottom of Figure 1 and near the bottom of Figure 2).

The `Puzzle` method for the example coding duel in Figures 1 and 2 is `public static int Puzzle(int x)`. The feedback given to the player on some selected input values is displayed as a table near the bottom of the screen (in Figure 2). A table row beginning with a check mark in a green circle indicates that the corresponding test is a passing test. Formally, the return values of the secret implementation and player implementation (i.e., the `Puzzle` method implementation) are

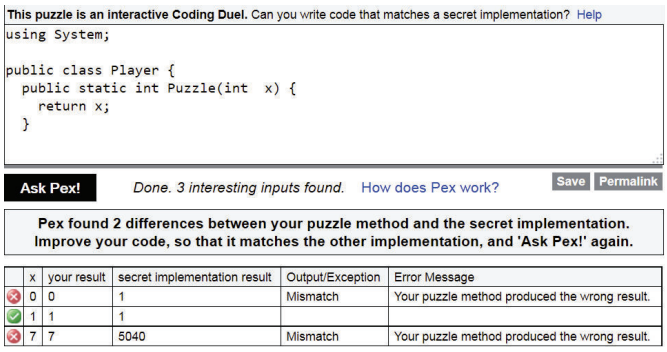

Fig. 2. The user interface of the Pex4Fun website

the same for the same test input (i.e., the `Puzzle` method argument value). A table row started with a red circle with a cross indicates that the corresponding test is a failing test: the return values of the secret implementation and player implementation are different for the same test input. In the table, the second column "x" indicates the test input. The third and fourth columns "your result" and "secret implementation result" indicate the return values of the player implementation and secret implementation, respectively. The last two columns "Output/Exception" and "Error Message" give more details for the failing tests.

To start with a simple coding duel, a player can do the following steps. Click an example coding duel from the Pex4Fun website; then the player can see a player implementation that does not do much. Click "Ask Pex!" to see how the player implementation differs from the secret implementation. Compare the player implementation's result to the secret implementation's result. Analyze the differences and change the code to match the secret implementation's results for all input values or as many input values as the player can. Click "Ask Pex!" again. Repeat this process until the player wins the coding duel (i.e., no failing tests being reported in the table by Pex) or cannot make any progress.

When a player has won the duel, the player can try other coding duels. if the player signs in to Pex4Fun with a Microsoft account (such as Outlook or Hotmail), Pex4Fun has a richer experience, in terms of tracking the player's progress. It can record how many attempts the player has made on a specific coding duel, how many coding duels the player tried to win, eventually won, and which ones the player created himself/herself. Pex4Fun also remembers the last program text that the player wrote for any particular coding duel.

Behind the scenes, coding duels leverage the technique of DSE described in Section III. Given a method `Player.Puzzle(x)` from a player (initially being a player implementation specified by the game creator) and a method `Secret.Puzzle(x)` from the game creator (being the secret implementation), Pex explores the following synthesized method `Driver`[1] with DSE [22] by treating such method as a parameterized unit test [25]–[27] (also shown on the bottom-left side of Figure 1):

```
public static void Driver(int x) {
  if (Secret.Puzzle(x) != Player.Puzzle(x))
    throw New Exception("Mismatch"); }
```

Note that DSE would attempt to generate tests to cover all feasible branches (in fact all feasible paths) including the exception-throwing branch in the synthesized method if this branch is feasible to cover; covering this branch indicates behavior mismatches between the player implementation and the secret implementation. In particular, with DSE, Pex generates a test suite that is customized to both `Puzzle` methods. Each time the player submits a new implementation version of the `Player.Puzzle(x)` method by clicking "Ask Pex!", Pex generates a new test suite, showing any behavior mismatches to the secret implementation, or, if there are no mismatches, indicating that the player wins the coding duel.

V. Design Goals and Principles

During our iterative process of evolving and improving the design of the concept of coding-duel games, we derived design principles for coding duels. We started by observing user behaviors on earlier versions of coding-duel games. From that, two major goals emerged: engage players (e.g., blend in "fun") and enable better teaching and learning.

Based on our experiences, we derived five main design principles related to the goal of engaging players.

- The games need to be interactive and the interactions need to be iterative and involve multiple rounds. The players' interactions with the games (e.g., modifying the player implementation) should not be just one round of interaction.
- The feedback of the games given to the players should be adaptive. When a player makes a move (e.g., modifying the player implementation) in one iteration of interactions, the feedback given should be adaptive with respect to previous moves, rather than always giving the same feedback.
- The feedback given to the players should be personalized. This design principle is closely related to the previous one on being adaptive. Basically, different players very likely make different modifications on the player implementation. Our feedback should be based on the modifications made by the current player.
- The games should have a clear winning criterion. There should be no ambiguity for a player in terms of understanding and assessing the winning criterion. Criteria such as "having a good code structure" would be too vague.
- There should be no or few opportunities for the players to cheat the games. The design of the games and supporting platform should allow no or few cases where the players do not satisfy the winning criterion but manage to win the game anyway.

With respect to the second goal of enabling better teaching and learning, we make the design of the concept of coding

[1]For some return types of `Puzzle` such as `int[]`, the comparator `==` would be replaced with the corresponding deep equality comparator for the return type.

duels to help train different skills of the players, including but not limited to the following ones:

Induction skills. The displayed list of selected argument values (which exhibit different behaviors or same behaviors of the two implementations) are just sample argument values, i.e., they are not exhaustive argument values that exhibit different or same behaviors. Before figuring out how to change a player implementation to get closer to the secret implementation, the player needs to generalize from the observed sample values and the behaviors exposed by them.

Problem-solving or debugging skills. Solving a coding duel requires the player to conduct iterations of trials and errors. Based on the observed sample argument values and behaviors, the player needs to decompose the problem: grouping sample arguments that may exhibit the same category of different behaviors, e.g., due to lacking a branch with the conditional of `if (x > 0)`. Next the player needs to come up with a hypothesized missing or corrected piece of code that will make failing tests pass and passing tests still pass. Then the player needs to conduct an experiment to validate the hypothesis by clicking "Ask Pex!". Solving a non-trivial coding duel can involve exercising a range of different problem-solving skills.

Program-understanding and programming skills. If the initial player implementation is not that "dumb" and includes non-trivial code, the player needs to understand first what the player implementation is meant to do. It is obvious that the player needs to have good programming skills to solve a non-trivial coding duel.

Testing skills. Currently, Pex4Fun does not provide user interfaces to explicitly request the player to specify or control test-input values. These are always generated by Pex for the player implementation and secret implementation, respectively. However, the player has the capability of "controlling" what additional test-input values are displayed in the input-output table by adding additional branches to the beginning of the player implementation. For example, if the player adds `if (x == 10) throw New Exception();`, Pex would generate a row with the test-input value `x` as `10` (due to the nature of Pex in generating a new test-input value for covering a not-yet-covered branch). In this way, the player's testing skills get trained. In fact, as presented in Section VII-B, coding-duel exercises can be carefully designed to train students' testing skills on writing parameterized unit testing [25]–[27].

Specification-writing skills. Pex4Fun already includes a number of coding duels that serve to test and train specification-writing skills. In such a coding duel, the secret implementation includes code contracts [4], such as method preconditions and postconditions. Then the player is given a player implementation with the same implementation as the secret implementation except without the (full) written code contracts. The player is asked to write the missing code contracts based on feedback. In this way, the player's specification-writing skills get trained. Furthermore, as presented in Section VII-A, coding-duel game exercises can be carefully designed to train students' skills on writing formal specifications based on the given natural-language requirements.

VI. Teaching with Coding Duels

Duel construction. Any teacher or user can create and submit new coding duels, which other players can try to win. There are five simple steps for the user to follow.

- Step 1: sign in, so that Pex4Fun can maintain coding duels for the user.
- Step 2: write a secret implementation starting from a puzzle template where the user can write the secret implementation as a `Puzzle` method that takes inputs and produces an output.
- Step 3: create the coding duel by clicking a button "Turn This Puzzle Into A Coding Duel" (appearing after the user clicks "Ask Pex!").
- Step 4: edit the player implementation (i.e., program text visible to players) by clicking the coding duel Permalink URL, which opens the coding duel, and by filling in a slightly more useful outline of the implementation (with optional comments) that players will eventually complete.
- Step 5: publish the coding duel after the user finishes editing the visible `Puzzle` method text by clicking the "Publish" button.

A `Puzzle` method can be turned into a coding duel only if it fulfills certain requirements. In particular, it must have a non-void return type, so that the behavior of the secret implementation and the player implementation can be compared using their return values. The `Puzzle` method must have at least one parameter, so that Pex can generate argument values for it.

The game creator has great flexibility to control the difficulty of solving a coding duel by varying (1) the complexity of the secret implementation; (2) the similarity level of the player implementation (visible to players) to the secret implementation; (3) the strength of the hints given in code comments in the player implementation. These advantages in creating coding duels make Pex4Fun an attractive open platform for the community to contribute coding-duel games, besides the list of built-in coding duels created by us. Note that a teacher does not have to provide any test cases, as Pex can dynamically compute specialized tests that represent different and same behaviors between the secret implementation and a student's player implementation.

Virtual classroom. A teacher can integrate Pex4Fun in various ways: (1) use built-in coding duels and request the students to submit to the teacher their attempts to these coding duels; doing so allows to leverage the built-in collection of coding duels and the gaming experience; (2) reuse existing courses; and (3) create new course materials.

A course in Pex4Fun consist of pages, which contain text and code written by an author. Pages are written in a simple, text-based markup language. A teacher can combine existing pages into a course. The pages might have been written by the teacher or by any other author. The teacher invites students to

the course by sharing a registration link with them. A course can have multiple teachers.

Any user can become a student by registering for a course through the registration link. The student can then work through the pages that are part of the course. To pass the course, the student completes exercises in the form of coding duels. A student can unregister from a course at any time. A student can give read access to a teacher. The teacher is then able to monitor the progress of the student on every coding duel, including inspecting the playing history of the student on the coding duel. Such feature is valuable for the teacher to "replay" and "diagnose" the student's thinking process.

Social dynamics. To better engage users, we have developed a number of features related to social dynamics, making games in Pex4Fun a type of social games. For example, Pex4Fun allows a player to learn what coding duels other people were already able to win (or not). For a given coding duel opened by a player, the description-text box above the working area shows some statistic such as "Can you write code that matches a secret implementation? Other people have already won this Duel 477 times!".

The platform also allows ranking of players and coding duels. A player can click the "Community" link on the Pex4Fun main page to see how the player's coding-duel skills compare to other people. In the community area, there are high score lists, as well as coding duels that other people have published.

After winning a coding duel, a player can rate it as "Fun", "Boring", or "Fishy". All ratings are shared with the community. Players earn points for rating duels, and can go back and rate them later as well.

VII. Example Coding-Duel Exercises in Software Engineering Course

In this section, we present two groups of coding-duel exercises created for a graduate course on software engineering. The first group of exercises is for the topic of requirements, particularly on formalizing natural-language requirements to machine-checkable formal specifications. The second group of exercises is for the topic of testing, particular on writing parameterized unit tests [25]–[27]. The course page including the described exercises in this section can be accessed at http://pexforfun.com/gradsofteng. To access the course page, one needs to log in to Pex4Fun by clicking the "Sign In" on the top-right corner of the Pex4Fun website, and entering her Microsoft account (e.g., an Outlook or Hotmail account).

A. Requirements Exercises

We have designed requirements exercises to (1) train students to formalize natural-language requirements for machine-checkable formal specifications, and (2) allow students to learn the importance on avoiding writing ambiguous low-quality natural-language requirements. Before conducting the designed exercises, the students attended lectures given by the teacher on the topics of code contracts [4] and requirements engineering. In addition, the students were asked to go through learning materials on code contracts (which are already provided at the Pex4Fun website) but such task is optional. The students were given one week to finish the requirements exercises.

We have designed six coding-duel exercises including three in the low-difficulty level, one in the medium-difficulty level, and two in the high-difficulty level. In each exercise, the students are asked "Can you translate the natural-language requirement above the method blow to be in code contracts?" The coding-duel exercise in the medium level (the initial player implementation) is shown in Figure 3 and its solution (the secret implementation) is shown in Figure 4. We extracted the natural-language requirements (used in the exercises) from real-world evaluation subjects (API documents) used in related previous work [17]. For each of the three exercises in the low-difficulty level, only one code contract is needed and it is straightforward to translate the requirement to the code contract. For the exercise in the medium-difficulty level, four code contracts are needed and it is not difficult to translate the requirement to the code contracts, as shown in Figures 3 and 4. For the two exercises in the high-difficulty level, five and nine code contracts are needed, respectively. For each of the two high-difficulty exercises, two code contracts require the use of non-trivial regular-expression matching:

```
Regex.IsMatch(name.Substring(0,1),@"[a-z]")
Regex.IsMatch(name,@"^[a-z0-9]*$")
```

These two method invocations reflect the requirement sentence "This name also needs to be a valid identifier, which is no longer than 32 characters, starting with a letter (a-z) and consisting of only small letters (a-z), numbers (0-9) and/or underscores."

For one of the high-difficulty exercises, we intentionally included one requirement sentence that cannot be expressed as a code contract: "param:name:Name of this new object type. This name needs to be unique among all object types and associations defined for this application." This sentence is from a real-world API document. In our solution (secret implementation), we have no line of code contract corresponding to this requirement sentence. Having the students go through this exercise allows them to realize that not all real-world requirements can be formalized. Indeed, there were a few students double-checking with the teacher on whether it was fine not to formalize this requirement sentence (although they already won the coding duel).

The comments in the player implementation inform the students *completely* exactly what they need to accomplish in terms of the secret-implementation functionality. The students do not have to "guess" based on the feedback given by Pex4Fun. Such design style is in contrast to many coding duels in Pex4Fun, where "guessing" is heavily involved and no or few hints are given to players. The testing exercises presented next incorporate some level of "guessing".

B. Testing Exercises

We have designed testing exercises to (1) train students on writing parameterized unit tests [25]–[27], (2) allow students

```
//param:pref_id:(0-201) Numeric identifier of this preference.
//param:value:(max. 127 characters) Value of the preference to set.
//      Set it to "0" or "" to remove this preference.
//Can you write preconditions in code contracts for the above natural-language requirements?
public static int Puzzle(int pref_id, string value)
{
    return pref_id;
}
```

Fig. 3. An example requirements exercise in the medium-difficulty level

```
public static int Puzzle(int pref_id, string value)
{
    Contract.Requires(pref_id >=0);
    Contract.Requires(pref_id <=201);
    Contract.Requires(value != null);
    Contract.Requires(value.Length <=127);
    return pref_id;
}
```

Fig. 4. The solution to the example requirements exercise in the medium level shown in Figure 3

to realize the importance on writing high-quality test oracles in the form of parameterized unit tests, and (3) appreciate how an automatic test-generation tool can be leveraged to generate high-quality test data. Before conducting the designed exercises, the students attended a lecture given by the teacher on the topics of basic software testing and parameterized unit testing. In addition, the students were asked to go through learning materials on parameterized unit testing (which are already provided at the Pex4Fun website) but such task is optional. The students were given one week to finish the testing exercises.

We designed three coding-duel exercises, all in the same difficulty level. In particular, in each exercise, the students are asked "For each coding duel, you need to complete the given incomplete parameterized unit test to match the secret parameterized unit test for testing the `UBIntStack` class that implements a bounded stack that holds unique integer elements." In the lecture on parameterized unit testing, the teacher used this `UBIntStack` class as an illustrative example to engage the students to do in-lecture exercises for writing traditional unit tests (i.e., those without parameters) and how to generalize such traditional unit tests to parameterized unit tests [23].

One of the three coding-duel exercises is shown in Figure 5 and its solution (the secret implementation) is shown in Figure 6. In particular, in the player implementation given to the students, we already include test-scenario setup (including some assumptions) and test oracles (i.e., assertions), and the students are asked to fill in the middle part of the parameterized unit test, which is the `Puzzle` method itself. As shown in Figure 6, the middle part includes additional test-scenario setup (including some assumptions) and the method under test (i.e., the `Push` method).

For the testing exercises, the comments in the player implementation inform the students *partially* what exactly they need to accomplish in terms of the secret-implementation functionality. Thus the students need to do some "guessing" based on the feedback given by Pex4Fun. For example, the comments in the player implementation do not inform the students which method of `UBIntStack` is the method under test (it is `Push`) or what additional test-scenario setup is needed. In the example exercise in Figure 5, additional test-scenario setup includes the `elem` being pushed is not in the stack already and the stack is not full. The students need to "guess" such information based on the feedback given by Pex4Fun.

VIII. Classroom Experiences

Pex4Fun was adopted as a major platform for assignments in a graduate software engineering course (with more than 50 enrolled graduate students). The Pex4Fun course page is at http://pexforfun.com/gradsofteng. The course content is organized by different phases in software development life cycle: requirements, design, implementation, testing, and maintenance, etc. Various exercises are designed in the form of coding duels. We next illustrate the classroom experiences through the two example sets of exercises described in Section VII.

Overall class performance. For each course at Pex4Fun, the teachers of the course can view the status of the students in terms of solving the coding-duel exercises included in the course (in the form of coding duels being embedded in a course page included in the course). The status page is similar to the one shown in Figure 7. Each row in the status table corresponds to the status of a student. The table cells in the columns show the number of attempts tried by each student, with a green table cell indicating that the corresponding coding duel was solved by the student, a red table cell indicating that the corresponding coding duel was not solved by the student yet, and a blank table cell indicating that the corresponding coding duel was not yet attempted by the student. We next illustrate some summaries of the class performance.

For the six requirements exercises, all students successfully solved all the six exercises except four students (who never attempted any of the six exercises, likely due to late enrollment or misunderstanding of the assignment requirements) and two students (who could not solve the last two exercises or the

```
public static string Puzzle(int[] elems, int capacity, int elem)
{
    if (capacity <= 0) return "Assumption Violation!";
    if (elems == null) return "Assumption Violation!";
    if (elems.Length > (capacity + 1)) return "Assumption Violation!";
    UBIntStack s= new UBIntStack(capacity);
    for (int i = 0; i < elems.Length; i++)
       s.Push(elems[i]);
    int origSize = s.GetNumberOfElements();
 //Please fill in below test scenario on the s stack
 //including necessary assumptions (no additional assertions needed)


 //The lines below include assertions to assert the program behavior
    PexAssert.IsTrue(s.GetNumberOfElements() == origSize + 1);
    PexAssert.IsTrue(s.Top() == elem);
    PexAssert.IsTrue(s.IsMember(elem));
    PexAssert.IsTrue(!s.IsEmpty());
    return "s.GetNumberOfElements():" + s.GetNumberOfElements().ToString() + "; "
      + "s.Top():" + s.Top().ToString() + "; "
    + "s.IsMember(elem):" + s.IsMember(elem).ToString() + "; "
     + "s.IsEmpty():" + s.IsEmpty()  + "; ";
}
```

Fig. 5. An example testing exercise for testing a bounded integer stack

```
public static int Puzzle(int pref_id, string value)
{
    ....
 //Please fill in below test scenario on the s stack
 //including necessary assumptions (no additional assertions needed)
    if (s.IsMember(elem)) return "Assumption Violation!";
    if (s.GetNumberOfElements() >= s.MaxSize()) return "Assumption Violation!";
    s.Push(elem);
    ....
}
```

Fig. 6. The solution to the example testing exercise shown in Figure 5 for testing a bounded integer stack

last exercise, respectively). The smallest numbers of attempts to successfully solve the six exercises are 1, 1, 1, 1, 2, 2, respectively. The largest numbers of attempts to successfully solve the six exercises are 15, 7, 32, 72, 67, 33, respectively.

For the three testing exercises, all students successfully solved all the three exercises except one student (who could not solve any of the three exercises). The smallest numbers of attempts to successfully solve the three exercises are 3, 1, 1, respectively. The largest numbers of attempts to successfully solve the three exercises are 122, 40, 36, respectively.

Grading. The grading scheme is simple[2]: the teacher gave a student full credit for successfully solving an exercise and zero credit otherwise, no partial credit being given. The number of attempts used to successfully solve an exercise was not taken into account for grading. When the deadline for an assignment was reached, the teacher gathered the online status of the students in solving the assigned exercises and translated the percentage of exercises being successfully solved by a student into the assignment grade for the student. The grading was very efficient and is expected to scale to much larger classes at universities and MOOCs. Other grading criteria are possible.

[2]Note that other grading schema could be adopted. For example, extra credits can be given to those students who can successfully solve the exercise within top N in the student ranking based on the number of used attempts.

Exercise-solving process. The exercise-solving status of a student (similar to the one shown in Figure 7) additionally includes in-depth details of the student's exercise-solving process. When clicking the number of attempts made by the student, the teacher can inspect the list of historical versions of player implementations submitted by the student over time. The benefits of inspecting such historical versions by the teacher have been substantial in two main aspects. First, the teacher can get a quick understanding of how students are doing way before the deadline. For example, for the testing exercises, on the second day of the one-week exercise-performing period, the teacher found that several students (who already started working on the exercises) attempted for a large number of times to tackle the exercises by filling in very sophisticated code logic but failed. The teacher sent out an email to the whole class to give them additional hints on avoiding going for this unfruitful direction. Second, the teacher can get deep insights on the thinking and problem-solving process of a student by inspecting such historical versions. Such deep insights can be used by the teacher to provide customized guidance to those students who faced difficulties on specific exercise topics. For example, by inspecting the historical versions of sampled students who could not successfully solve the last two requirements exercises and the

three testing exercises, the teacher realized that some of these students had difficulties in formulating relatively complex regular expressions and some of these students had difficulties to understand the assumption concept in parameterized unit testing.

Summary. The classroom experiences with using Pex4Fun were very encouraging. As discussed earlier, from the perspective of the teacher, the benefits of leveraging Pex4Fun in classroom teaching could be substantial: reducing grading efforts, exposing students' learning process (beyond their final exercise solutions) along the way, etc. Although we have not yet conducted a formal study in assessing students' learning effectiveness, based on informal interactions with students, we found that such in-time iterative feedback provided by Pex4Fun allowed students to have guidance along the way. Even when assignment requirements were clearly given just like other traditional exercises, such guidance allowed students to improve their working solutions (if not correct) and the students had more confidence in the correctness of their final solutions (judged by Pex4Fun). For the exercises with "guessing" elements such as the testing exercises, the exercise-solving process of the students had the fun aspect, allowing students to search for both the requirements and the solutions in the problem/solution space. In future work, we plan to conduct some formal evaluation on students' learning when conducting Pex4Fun exercises.

IX. Open-Contest Experiences

Holding a contest of solving coding duels in either a public setting or a classroom setting can serve the purpose of engaging students to solve coding duels in a dynamic social context within a specific period of time. In May 2011, Microsoft Research hosted a contest on solving coding duels (http://research.microsoft.com/ICSE2011Contest) at 2011 International Conference on Software Engineering (ICSE 2011). During the main ICSE program, conference attendees could register a nickname at Pex4Fun and complete as many coding duels as possible within the ICSE 2011 main conference period. Whoever solved the most coding duels by the end of the period won the contest.

The ICSE 2011 coding-duel contest received 7,000 Pex4Fun attempts, 450 duels completed, and 28 participants (though likely more, since some did not actually enter the official ICSE 2011 course to play the coding duels designed for the contest). Through initial investigation of the registered participants, we suspect that these 28 participants were mostly graduate students. Among the 28 participants, 4 participants completed all 30 duels, with many more completing all except for the pesky duel #13. Figure 7 shows the status table of the ICSE 2011 coding-duel contest[3]. The first column of the table shows the nicknames of the participants. Columns 2-31 correspond to the 30 coding duels designed for the contest, with the same color coding as described in Section VIII.

[3]The status table in Figure 7 includes 29 entries because one new participant registered the ICSE 2011 course and attempted exercises there after the ICSE 2011 period.

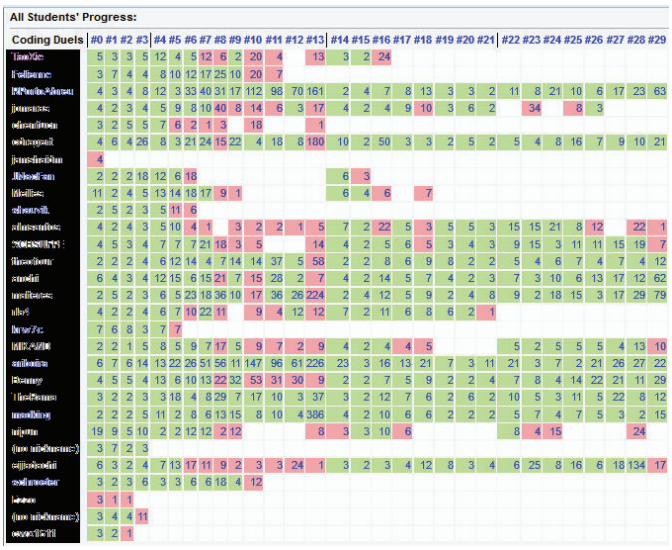

Fig. 7. The status table of the ICSE 2011 coding-duel contest

From the table, we could observe that only 4 participants successfully solved duel #13, with the number of attempts as 37, 161, 226, and 386, respectively. This duel #13 is the most difficult one among the 30 duels. The second most difficult duel is duel #10, which only 6 participants successfully solved, with the number of attempts as 4, 8, 14, 17, 112, and 147, respectively. Overall, difficult duels tend to have more varieties on the numbers of attempts made to solve the duels than easy duels (e.g., duels #0-#3). It might be that different participants who solved a difficult duel may have prior knowledge on the domain of the duel such as a particular mathematical problem. To some extent, the solving process is a searching process in the space of all possible implementations. A participant's prior knowledge in the search space helps speed up the searching process. In future work, we plan to conduct detailed manual analysis of solving histories of participants to investigate how and why participants succeeded or failed to solve a duel. We expect that such analysis results would provide insights on how to more effectively design coding duels with respect to different learning or problem-solving styles besides different skills.

X. Conclusion

In typical large courses at universities and especially MOOCs, assignment grading is a critical factor for their success in teaching and learning effectiveness. Traditional ways of assignment grading are not scalable and would typically not give in-time or interactive feedback to students to engage them in completing the assignments. To address these issues, in this paper, we have presented an interactive-gaming-based teaching and learning platform called Pex4Fun for programming languages such as C#, Visual Basic, and F#. It is a browser-based teaching and learning environment with target users as teachers and students for introductory to advanced programming or software engineering topics. The platform also provides various features to engage students in

a social learning environment. Teachers can enjoy the benefits of automated grading of exercises assigned to students, making the platform applicable in the context of large courses at universities and MOOCs.

Pex4Fun was adopted as a major platform for assignments in a graduate software engineering course, and a coding-duel contest was held at ICSE 2011 for engaging conference attendees to solve coding duels in a dynamic social contest. Pex4Fun has been gaining high popularity in the community: since it was released to the public in June 2010, the number of clicks of the "Ask Pex!" button (indicating the attempts made by users to solve games at Pex4Fun) has reached over one million (1,135,931) as of March 3, 2013.

In future work, we plan to study Pex4Fun's effect on engaging players across different genders, ages, skills, etc. We also plan to conduct detailed manual analysis of recorded logs of students' duel-solving process to identify the students' problem-solving and learning styles. We expect that insights gained in this analysis can help better design the teaching and learning materials including the design of specific coding duels for maximizing the learning outcomes.

We plan to explore the direction of adapting previous work on program synthesis [11] to automatically synthesize coding duels given generic specifications of the secret implementation and player implementation in a coding duel to be developed (e.g., the complexity level of the implementations, and the extent of behavioral differences between the two implementations, the types of syntactic differences between the two implementations). Such automatic synthesis of coding duels for a programming concept can be valuable because a student may need different coding duels (with different secret implementations) for the same programming concept to reinforce the student's learning of the programming concept. Manually providing such a long list of coding duels may not be feasible.

We plan to explore intelligent-tutoring support [13], [19], [20] in the Pex4Fun platform. The current platform provides hints and assistance to the students only in the form of failing and passing tests. When a student cannot make progress in solving a coding duel, the current platform does not give additional "hand holding". We plan to incorporate intelligent-tutoring support by mining a large number of other students' solving logs for the same coding duel and analyzing statically and dynamically the differences between the secret and player implementations.

Acknowledgment. Tao Xie's work is supported in part by NSF grants CCF-0845272, CCF-0915400, CNS-0958235, CNS-1160603, an NSA Science of Security Lablet grant, a NIST grant, a Microsoft Research Software Engineering Innovation Foundation Award, and NSF of China No. 61228203.

REFERENCES

[1] Coursera. https://www.coursera.org/.

[2] Coursera MOOC on Software Engineering for SaaS. https://www.coursera.org/course/saas.

[3] EdX. https://www.edx.org/.

[4] Microsoft Research Code Contracts. http://research.microsoft.com/projects/contracts.

[5] Udacity. http://www.udacity.com/.

[6] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.

[7] S. H. Edwards and M. A. Perez-Quinones. Web-CAT: automatically grading programming assignments. In *Proc. Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, pages 328–328, 2008.

[8] A. Fox and D. Patterson. Crossing the software education chasm. *Communications of the ACM*, 55(5):44–49, May 2012.

[9] S. Freeman and C. Parks. How accurate is peer grading. *CBE– Life Sciences Education*, 9:482–488, 2010.

[10] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005.

[11] S. Gulwani. Dimensions in program synthesis. In *Proc. International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 13–24, 2010.

[12] D. Jackson and M. Usher. Grading student programs using ASSYST. In *Proc. SIGCSE Technical Symposium on Computer Science Education (SIGCSE)*, pages 335–339, 1997.

[13] W. Jin and A. Corbett. Effectiveness of cognitive apprenticeship learning (CAL) and cognitive tutors (CT) for problem solving using fundamental programming concepts. In *Proc. SIGCSE Technical Symposium on Computer Science Education (SIGCSE)*, pages 305–310, 2011.

[14] K. Jordan. HCI – interesting issues with peer grading, 2012. http://moocmoocher.wordpress.com/2012/07/18/hci-interesting-issues-with-peer-grading/.

[15] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[16] K. Masters. A brief guide to understanding MOOCs. *The Internet Journal of Medical Education*, 1, 2011.

[17] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language API descriptions. In *Proc. International Conference on Software Engineering (ICSE)*, pages 815–825, 2012.

[18] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, 2005.

[19] S. C. Shaffer. Ludwig: an online programming tutoring and assessment system. *SIGCSE Bull.*, 37:56–60, June 2005.

[20] L.-K. Soh. Incorporating an intelligent tutoring system into CS1. *SIGCSE Bull.*, 38:486–490, March 2006.

[21] J. Spacco, D. Hovemeyer, W. Pugh, J. Hollingsworth, N. Padua-Perez, and F. Emad. Experiences with Marmoset: Designing and using an advanced submission and testing system for programming courses. In *Proc. Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, pages 13–17, 2006.

[22] K. Taneja and T. Xie. DiffGen: Automated regression unit-test generation. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 407–410, 2008.

[23] S. Thummalapenta, M. Marri, T. Xie, N. Tillmann, and J. de Halleux. Retrofitting unit tests for parameterized unit testing. In *Proc. International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 294–309, 2011.

[24] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *Proc. International Conference on Tests and Proofs (TAP)*, pages 134–153, 2008.

[25] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proc. joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 253–262, 2005.

[26] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Softw.*, 23(4):38–47, 2006.

[27] T. Xie, J. de Halleux, N. Tillmann, and W. Schulte. Teaching and training developer-testing techniques and tool support. In *Proc. Annual ACM Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH), Educators' and Trainers' Symposium*, pages 175–182, 2010.

[28] A. Zeller. Making students read and review code. In *Proc. Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, pages 89–92, 2000.