Experience with Constructing Code Hunt Contests

R. Nigel Horspool
University of Victoria
Victoria, Canada
nigelh@uvic.ca

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

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

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

ABSTRACT

Puzzles are the basic building block of Code Hunt contests. Creating puzzles and choosing suitable puzzles from the puzzle bank turns out to be a complex operation requiring skill and experience. Constructing a varied and interesting mix of puzzles is based on several factors. The major factor is the difficulty of the puzzle, so that the contest can build up from easier puzzles to more difficult ones. For a successful and fun contest aimed at the expected abilities of the contestants, other factors include the language features needed to solve the puzzle, clues to provide when the puzzle is presented to the player, and test cases to seed into the Code Hunt engine. We describe our experience with contest construction over a period of year and provide guidelines for choosing and making adjustments to the puzzles so that a Code Hunt contest will provide a satisfying trouble-free experience for the contestants.

Categories and Subject Descriptors

K.3.1 [**Computers and Education**]: Computer Uses in Education – *computer-assisted instruction*; K.8.0 [**Personal Computing**]: General – *games*.

General Terms

Languages.

Keywords

Programming contests, Code Hunt game, Unit tests

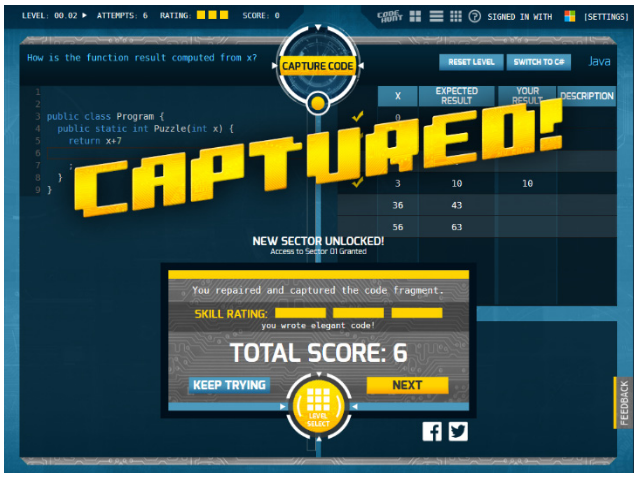
1. INTRODUCTION

Code Hunt is a game for coding against the computer by solving a sequence of puzzles of increasing complexity. Code Hunt runs in any modern browser at http://www.codehunt.com. The game is structured into a series of sectors, which in turn contain a series of levels. In each level, the player must write code that implements a particular formula or algorithm. As the code develops, the game engine gives custom progress feedback to the player, generated by the testing engine, Pex [6]. It is part of the gameplay that the player learns more about the nature of the goal algorithm from the progress feedback.

The player can write code in an editor window, using either C# or Java as the programming language. This code must implement a top-level function called `Puzzle`. The puzzle has some input parameters, and it returns a result. The player tests if the current code implements the goal algorithm: by pressing on a big "CAPTURE CODE" button shown in Figure 1 and Figure 2.

The result is either a compilation error, or a list of mismatches and agreements with the goal algorithm. Figure 2 shows the code on the left, and the mismatches (red crosses) and agreements (yellow checkmarks) are shown on the right.

If the code compiles and there are no mismatches the player wins this level – or as the game puts it, the player "CAPTURED!" the code, as shown in Figure 1. A "skill rating" is assigned to the player's code, reflecting the elegance of the solution, measured by its succinctness (a count of instructions in the compiled .NET intermediate language).



How is the function result computed from x?

```
public class Program {
  public static int Puzzle(int x) {
    return x+7
      ;
  }
}
```

X	EXPECTED RESULT	YOUR RESULT	DESCRIPTION
0			
3	10	10	
36	43		
56	63		

You repaired and captured the code fragment.
SKILL RATING:
you wrote elegant code!
TOTAL SCORE: 6

Figure 1 After solving a puzzle, the player gets a score

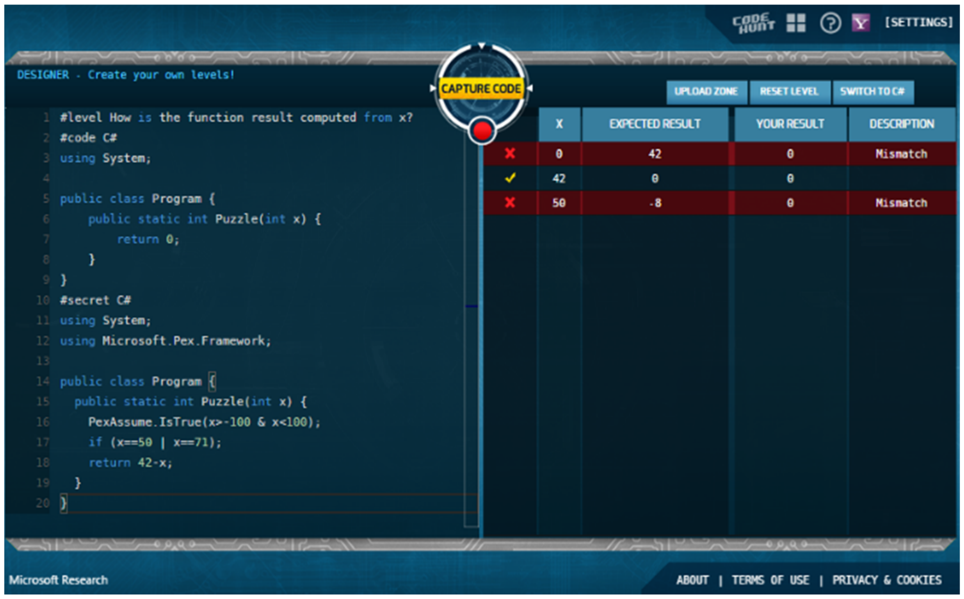
The default game provided by Code Hunt has a theme for each sector. The general idea is for a new programming construct to be needed when solving puzzles in the next sector. For example, one sector may need if-statements in the solutions, while the next sector may require singly nested loops. The intention of this arrangement was for Code Hunt to be used as an educational tool, giving students experience with programming language features one-by-one. Although the educational use of Code Hunt continues, we have been using Code Hunt to create programming contests where the numbers of simultaneous contestants online sometimes reach into the thousands.

The Microsoft Code Hunt team have, to date, constructed and run over 30 contests around the world. Each contest typically contains 12 to 36 puzzles. The contests are organized as a series of two or more sectors, each sector normally containing four to six puzzles. The contests run on the Code Hunt website but are accessed through specially protected URLs.

CHESE'15, July 14 2015, Baltimore, MD, USA
© 2015 ACM. ISBN 978-1-4503-3711-3/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2792404.2792405

In setting up a contest, we try to organize the puzzles so that there is a gentle increase in difficulty level as players advance from one sector to the next. Within a sector, we try to choose puzzles which are at approximately the same level of difficulty. However a puzzle which is being used in a contest for the first time may prove to be unexpectedly challenging. For that reason, we allow a player to progress to the next sector with one puzzle left unsolved in the current sector.

In previous work [1], we discussed our experience with contests and many of the results we have obtained. This short experience paper is intended to provide advice on how to choose and fine-tune the puzzles when a new contest is being created.



```
#level How is the function result computed from x?
#code C#
using System;

public class Program {
    public static int Puzzle(int x) {
        return 0;
    }
}
#secret C#
using System;
using Microsoft.Pex.Framework;

public class Program {
  public static int Puzzle(int x) {
    PexAssume.IsTrue(x>-100 & x<100);
    if (x==50 | x==71);
    return 42-x;
  }
}
```

	X	EXPECTED RESULT	YOUR RESULT	DESCRIPTION
✗	0	42	0	Mismatch
✓	42	0	0	
✗	50	-8	0	Mismatch

Figure 2 Using the Code Hunt Designer Tool

2. PUZZLE SELECTION

As of today, we have a puzzle catalog containing 391 distinct puzzles plus another 44 puzzle clones. (We consider a clone to be a duplicate of an existing puzzle but with different values for some internal parameters.) Nearly all these puzzles have been previously used in one or more contests, providing us with some data about their difficulty levels. We use the average number of tries that a player had at solving the puzzle as our difficulty indicator. It must only be viewed as a rough guide because the pools of contestants do not necessarily possess similar abilities. The average number of tries currently ranges from a high of 40.8 (for a puzzle based on a function used in number theory) to a low of 1.1 (for a puzzle where the solution is the expression `-x`).

Some puzzles have not yet been used in a contest. These puzzles plus any newly created ones have only subjective difficulty ratings. These are difficulty ratings provided by the puzzle creator and range from 1 (easy) to 5 (extremely challenging). The subjective ratings must be used with caution however, because we have been surprised many times when a puzzle thought to be easy turns out to be much more challenging when used in a contest. Our most extreme example of such a mismatch is a puzzle with a subjective rating of 2 yet it required 14.2 tries on average to be solved. (This is a puzzle where the input comprises the numerator and denominator of a rational fraction, and the result is those numbers with common factors removed.) Fortunately severe mismatches are becoming rarer as we gain more experience.

Within the sector, we normally select puzzles with similar difficulty ratings and which provide variety. Our puzzles have a descriptor which can be one of *numbers*, *bools*, *binary*, *string*, or *array*. It crudely describes the dominant datatypes manipulated by the puzzle solution. We choose puzzles which span these different datatypes as much as possible. We feel that variety is needed to maintain interest amongst the players, and to avoid favoring or disadvantaging players who possess uneven programming skills.

Choosing the puzzles for each sector can be an iterative process because these puzzles need to be tried out and tuned, as explained below. We often discard a puzzle instead of laboring through making adjustments to the puzzle to make it more usable in the contest.

3. SETTING UP A PUZZLE

When a puzzle has been selected as a possible candidate for use in the contest, we paste that puzzle's code into the Designer tool of Code Hunt (reached from the Settings tab). Clicking the 'Capture Code' button will immediately show us if there are any compilation errors. If there are no compilation errors, Code Hunt will display the initial set of test cases that would be shown to a contestant if the initial code template is submitted as a solution. A screenshot of the Designer tool in use is shown in Figure 2. The puzzle developer has just pasted the specification for the puzzle in the left half of the window and clicked on the Capture Code button. That action caused the table of test cases to appear in the right half of the window.

Compilation errors are only likely when a new puzzle is being developed since nearly all puzzles in the catalog have been used in previous contests. With all but the simplest puzzles, it is valuable to edit the initial code template and see how the test cases change as different partial or incorrect solutions to the puzzle are submitted.

All puzzles, especially new puzzles, need to be checked for the following issues.

1. Does arithmetic overflow occur with some of the test cases?
2. Are `bool` values provided as test inputs restricted to just the C# implementations of true and false?
3. Are test inputs and test results displayed in a readable format?
4. Is the contestant given enough clues so as to make the puzzle reasonable?
5. Does the Pex engine generate a reasonable selection of test inputs?

We elaborate on each of these issues below.

3.1 Overflow

The Pex engine used inside Code Hunt to generate the test cases is trying to 'break' the code. Sometimes it will generate inputs which cause overflow. For example, a puzzle whose secret solution is:

```
public static int Puzzle(int X, int Y) {
   return Math.Abs(X*Y);
}
```

causes Code Hunt to display the following test results:

X	Y	Expected Result
0	0	0
838	646	541348
1115695448	1073760398	289952048
1	Int.MinValue	null

For the test cases shown in the third and fourth rows, overflow has occurred. The overflow for the third row occurred with the multiplication operation. That operation is not checked, and the result is an integer which has been truncated to 32 bits. The overflow for the fourth row occurs inside the `Abs` function, and this function threw an exception (`System.OverflowException`). Code Hunt displays `null` as the function result in this case, and the nature of the exception is reported below the table of results.

Unless the contestants are systems programmers, i.e. people who understand overflow issues well, constraints should be added to the secret solution to prevent overflow from occurring. For example, limiting both X and Y in this example to the range -100 to +100 would be reasonable.

3.2 Boolean Values

Although `bool` values are limited in both Java and C# to just `true` or `false`, the .NET runtime implements a `bool` as an unsigned byte which can contain integers from 0 to 255, where 0 represents `false` and anything else means `true`. The Pex engine sometimes generates alternate representations for `true`, which is unfortunate. For example, the puzzle whose secret solution contains just the one statement "`return X&&Y;`" produces these two test cases:

X	Y	Expected Result
false	false	false
true (0x02)	false	false

That is liable to be confusing to a player, and the secret solution needs to be augmented with constraints which limit the test inputs to the C# representations of `true` and `false`. A reliable way to implement such constraints in our example is to code the secret solution as follows:

```
public static bool Puzzle(bool X, bool Y) {
   bool T = 1>0;
   PexAssume.IsTrue((X==false | X==T) &
        (Y==false | Y==T));
   return X&&Y;
}
```

Note that if the constant `true` is used rather than the variable `T` in the constraints, Pex continues to generate non-standard true values for the test cases.

3.3 Readable Values

When a puzzle uses strings for an input or for its result, the Pex engine will often generate test cases where the strings contain non-printable characters. Such characters are displayed by Code Hunt with hexadecimal notation. For example, the string `"\0"` is often chosen as a test input and displayed in the test results table. To eliminate hexadecimal character codes from the inputs, more constraints should be added to the secret solution. Many string puzzles restrict strings to contain characters in just the 'a' to 'z' range; many others allow both upper case and lower case letters as well as spaces.

Another issue which affects readability is the display of array results. If the array is relatively long and/or contains elements which require several characters to be displayed (such as large integers or strings or subarrays), the narrow display columns on the webpage cause the array to wrap around over several lines. This should be avoided if possible; constraining the arrays to contain only a modest number of elements is desirable.

3.4 Providing Adequate Clues

Unless the puzzle is particularly easy, it is unfair to present the contestant with a puzzle without providing some modest clues as the nature of the function they have to discover. The clues can take several forms. Any or all of them can be, and have been, used in a contest puzzle. The possibilities include these.

3.4.1 Descriptive Argument Names

Choosing suggestive names for the arguments of the `Puzzle` method. For example, we might name an array argument `List`.

3.4.2 Helpful Puzzle Headings

Providing a helpful directive as the puzzle heading. The left column where the contestant edits the code has a heading which is provided by the puzzle creator. For example, one puzzle displays the message "How does this function transform the string?". Even though the function signature shows a string argument and a string-valued result, the extra information that the input string must be transformed will get the contestant thinking in the right frame of mind.

3.4.3 Code Comments

The puzzle creator can include detailed comments inside the solution template provided to the contestant. The comments might explain how the function arguments encode a particular data structure, or they might simply point the contestant in the right direction.

For example, the catalog contains a puzzle which almost everybody would find impossible. This puzzle takes its input argument x, an integer greater than zero, and counts how many times the operation: "if x is even then divide x by 2; otherwise multiply x by 3 and add 1" is repeated before x equals 1. That count is the function result. There is no known closed formula for the solution, though it is known that the computation will terminate for all input values implementable on a 64 bit computer. For this puzzle, the solution template contains the comment:

```
// Refs: http://www.numbertheory.org/php/
//           collatz.html
//        http://en.wikipedia.org/wiki/
//           Collatz_conjecture
```

This comment should cause the contestant to visit one or both webpages to discover that the finite nature of the number of iterations is a well-known conjecture in number theory [2].

It is entirely up to the puzzle designer as to what help can be provided in the form of comments. Some of the puzzles in the default Code Hunt zone are not puzzles at all because the comments tell the contestant exactly what must be coded as the solution.

3.4.4 Partially Completed Function Template

Nearly all puzzles in the catalog provide a minimal body for the puzzle function. That body consists of a single return statement, where the value being returned is typically a default value such as

0 or an empty string, whatever is appropriate for the function's result type. The puzzle creator can, instead, provide some initial code which is suggestive of the solution's structure or which provides help in other ways.

3.5 Generating Helpful Test Values

The Pex engine is biased towards generating simple test input values. It therefore preferentially chooses 0 for integer values, `'\0'` for character values, and `null` for string or array values.

In the absence of some extra work from the puzzle creator, the contestant might see test inputs for an argument whose type is array of strings with some or all of these values:

```
null, {}, {null}, {null, null}, {""},
{ "", null}, { "\0", null}, etc.
```

Such test values may be sufficient to show that the contestant has not provided the correct solution, but they will usually not provide much information about what computation is being performed by the secret solution.

The contestant can see more test input values by providing a series of if statements in their trial solution. For example, the trial solution might be coded as follows:

```
public static string Puzzle(string[] a) {
  if (a.Length==3 && a[0]=="abc" &&
      a[1]=="def" && a[2]=="ghi")
     return "123";
  return "";
}
```

That structure should cause Code Hunt to display what the expected result is when that particular array value is provided as the test input. However it is tedious for the contestant to construct such test cases. We believe that Code Hunt becomes more enjoyable if some non-trivial test inputs are generated without needing to be prodded by the player.

Most puzzles in the catalog have some test cases explicitly included in their secret solutions. These can be combined with constraints that serve further to exclude less interesting input values. To continue the previous example, the secret solution might include the particular array of strings as a test case with code like the following. (The Code Hunt Designer Manual [3] explains all the constructs used in this example and the reasons for coding it in this manner.)

```
public static string Puzzle(string[] a) {
  PexAssume.IsNotNull(a);
  PexAssume.IsTrue(a.Length>0&a.Length<5);
  for(int i=0; i<a.Length; i++)
    PexAssume.IsNotNullOrEmpty(a[i]);
  // provide control flow path
  if (a.Length==3 && (a[0]=="abc" &
      a[1]=="def" & a[2]=="ghi"))
     /* do nothing */ ;
  ...
  // remainder of function omitted
  return result;
}
```

Trying the puzzle oneself is the best way to see which test cases are generated by Code Hunt.

It should be noted that inserting the extra control flow paths does not guarantee that Code Hunt will actually display test cases corresponding to those paths. The reason is that Code Hunt limits how many paths will be explored and how much CPU time is expended on the analysis, and the particular path that provides the desired test case may not be included in the analysis. Sometimes moving the if-statement which specifies the desired test case to a different position in the secret solution will cause that test case to be generated. Some experimentation may be needed.

4. UPLOADING A CONTEST

When all the puzzles are ready, they are put together in order in a contest file. Additional commands in the file provide names for the contest and the sectors. Optional commands can define the times when the contest will be active. Full details are provided in the designer manual [3]. When the file is complete, its contents are copied into the Code Hunt Developer Tool window. Clicking on the Capture Code button causes the contest to be checked and uploaded to the cloud. In a short while, the system returns with a URL which is then the special access point for that contest's zone.

5. RELATED WORK AND PLANS

Programming contests have run for decades. A recent comprehensive survey of formats used for on-line programming contests and how such contests can be used in education and training has been provided by Combéfis and Wautelet [4].

A commonly used contest format is to have a problem specification, a reference solution and a set of test fixed cases [5]. In Code Hunt, we still write the reference solution, but the system generates test cases, and the specification is not given – that is part of the fun of the game.

We are currently working on curating all the puzzles into a portal, so that others, apart from ourselves, can create contests. We are also considering how to extend the website so that a contest can provide a spectator experience too. The idea is that spectators will be able to observe how contestants are faring and can see events such as when a contestant solves a puzzle in a new fastest time.

The demand for data on how people reach a solution in programming is high, so we have released all 13,000 programs from one contest at http://www.github.com/microsoft/code-hunt . We will soon release another data set.

6. ACKNOWLEDGMENTS

Our thanks to all members of the Code Hunt team for implementing this amazing website.

7. REFERENCES

[1] Judith Bishop, R Nigel Horspool, Tao Xie, Nikolai Tillmann, Code Hunt: Experience with Coding Contests at Scale, ICSE (JSEET Track), 398-497, 2015

[2] Collatz conjecture. Wikipedia. URL: http://en.wikipedia.org/wiki/Collatz_conjecture

[3] Code Hunt Designer Manual. URL: https://www.codehunt.com/docs/designer.html.

[4] Sébastien Combéfis, Jérémy Wautelet, Programming Trainings and Informatics Teaching Through Online Contests. Olympiads in Informatics, vol 8, 21-24, 2014.

[5] From Baylor to Baylor, lulu.com, by Miguel A. Revilla (Compiler), William B. Poucher (Foreword), 2010

[6] Tillmann, N., and de Halleux, J. Pex – White Box Test Generation for .NET. Proc. Tests and Proofs (TAP), pp 134–153, 2008.