Using a design recipe to teach JavaScript code

Ronald Kluft (838812818)

May 10, 2015
version 1.1

1 Introduction

Can a structured action plan be used to teach a student the JavaScript programming language in a distance learning environment?

In this paper the result of a short domain study on the usage of a recipe to teach a student how to design a program or function is presented. It is part of the JavaScript–examiner[1] project. This project is intended to create a web-based learning environment for a student to learn the JavaScript language, the subject of the course *Web applications: the Client Side* of the Open Universiteit (OU). In this course, the student is encouraged to use a design recipe which is mainly borrowed from the online course "Introduction to Systematic Program Design - Part 1" of the University of British Colombia, by prof. Gregor Kiczales [Kiczales, 2015], and based on chapter 3.1 of the book Designing Functions of the 2nd edition of the book "How to Design Programs" by Felleisen, Findler, Flatt and Krishnamurthi [Felleisen et al., 2015]. In this paper the second version of the book has been chosen, with writing in progress. References to the OU course, the online course and the book, will be noted, for brevity, as [WAC], [Coursera] and [HtDP] respectively.

In the description mainly the term function will be used, because this is mostly the structure that will be described, but the recipe isn't restricted to this structure.

In this paper the steps the recipe consist of will be described, and the possibility of usage of these steps in the JavaScript–examiner. The methodology of writing functional examples before writing the body of the function has also been looked into more deeply. This is called Test first Development (TfD) [Langr, 2001], Test Driven Development [Edwards, 2003], [Janzen and Saiedian, 2008], [Sommerville, 2011] or Test Driven Design (TDD) [Proulx, 2009]. In the context of learning, also the term Test Driven Learning (TDL) [Janzen and Saiedian, 2006], [Proulx, 2009], [Janzen and Saiedian, 2008] is used.

The checking of elegant JavaScript is discussed in [Arkenaar, 2014], and will not be part of this paper. A study on the feedback that will be given to the student has been performed by [Nieuwenhuize, 2014].

[1] https://github.com/Slotkenov/JavaScript-examiner/

The study of [Bieniusa et al., 2008] also mentions the fact that their students often copy code from each other, thus making them guilty of plagiarism. This paper won't go deeper into this, but may be subject of a later study, possibly after a fair amount of solutions have been submitted. The solutions can then be evaluated and a conclusion can be drawn.

1.1 Context

The JavaScript–examiner is a tool to examine submitted exercises from a student of the OU who participates in the course [WAC]. The purpose of the tool is two-folded : it will reduce the workload of the tutor, and let the student anonymously submit a solution to remove possible fear of having degraded by submitting wrong answers.

In chapter 6 of [WAC] the use of a design recipe is introduced. Here, a number of rules are given to which the student should apply, in order to use the recipe to learn to program in a stepwise manner, where the previous step is used in the following. Where [HtDP] and [Coursera] are focussing on the Beginning Student Language (BSL) of the Dr.Racket programming environment, [WAC] is focussing on JavaScript.

In the subsequent steps, the general concept will be explained and, if applicable, be matched with JavaScript. The intention is to learn the recipe with simple functions, and then use the recipe to design harder functions.

The JavaScript–examiner is not an interactive tool, that supplies feedback after every step. Rather, it accepts a complete solution from the student and, if necessary, gives proper feedback on the submitted code to help the student gain the required skills.

2 Recipe

The design recipe discussed in [HtDP] consists of 6 steps, where the version of [Coursera] has 5 steps.

There is a difference between the various recipes, which will be named if they occur. The recipe is like a top-down structure. The step at hand is helped by the step before, and helps with the step after it. This can be helpful in checking the recipe, because for instance, the names of the function has to be the same, and also the number of arguments. This can also help in determining the feedback given back to the student.

There is a difference between version 1 and version 2 of [HtDP]. In the first version the text mentions 5 phases, and in the accompanying table of that text, there are even as much as 4 phases (with Contract, Purpose and Header combined). In the second version of [HtDP], there are 6 phases. Here the 2nd version will be exemplified, because this is the latest.

1. Express how you wish to represent information as data.
2. Write down a signature, a purpose statement, and a function header.

3. Illustrate the signature and the purpose statement with some functional examples.

4. Take inventory, to understand what are givens and what we do need to compute.

5. Write executable expressions and function definitions.

6. Test the function on the examples that you worked out before.

The first item is not included in the [Coursera] version of the recipe [2], that consist of these five steps :

1. Signature, purpose and stub.

2. Define examples, wrap each in check-expect.

3. Template and inventory.

4. Code the function body.

5. Test and debug until correct.

In the following, the 6 steps of the [HtDP] version will be elaborated.

2.1 Express how you wish to represent information as data

This step is merely a one sentence summary of the exercise.

[Ramsey, 2015] splits this step into two parts: Data definition and Data example. His students often forgot to write data examples, and he states that "Data examples enjoy no comparable support."

Though it has no extra usage to the explanation of work of the student, one possibility of the JavaScript–examiner could be to check if it is a single sentence, but examination of the content could be overhead, and very hard to do, because there can be a great number of varieties.

2.2 Write down a signature, a purpose statement, and a function header

In this step the student has to give the data types that is consumed and produced by the function. It should reflect the number of arguments the function expects and what type it returns. In the first, simple functions, the data types will be the standard built-in data types. But as the functions get more complex, the data types can get more complex as well. In [WAC] (LE6, 1.6) the construction of more complex types is explained. The JavaScript–examiner could check the number of arguments and names of the data types. It should then be mentioned explicitly in the question that the arguments and return type should be given, because in JavaScript these can be omitted.

[2] https://class.coursera.org/programdesign-002/wiki/view?page=HtDFunctions

The exact names of the arguments don't have to match, but a check can be done on the number and the location in the signature.

The purpose statement should be a single line which states what the output of the function is, in terms of the input. Writing good purpose statements is very hard [Ramsey, 2015]. It is important that only a single sentence is given. Otherwise the function will have too much functionality and should be split into more functions. The JavaScript–examiner should test for the fact that it is only a single line. The submission of the student can also be saved, for checking against an other student's submissions, thus enhancing the tool. Reviewing the saved purpose statements may be a task of the tutor, to ensure that they are relevant. The saved statements may also be used later for the aforementioned plagiarism check. The purpose statement can be compared with the standard solution of the tutor. Although it can be very hard to give a detailed evaluation of the statement, because it is given in natural text.

The function header [HtDP] or stub [Coursera] is the actual header of the function to write. This should be exactly the same as the function header in the submitted code. This can be checked and saved by the JavaScript–examiner, to give feedback about the consistency of the header. Its purpose is to give the name of the function, as well as the type of data it expects as its input.

The tutor can supply a standard header for an exercise, but it can be hard to match it exactly, because the student may not use the exact same names for the function and its parameters. Again, only the presence of the names can be checked. Another option is to state in the exercise that the exact names given should be used. Then the checking of these names can be automated.

2.3 Illustrate the signature and the purpose statement with some functional examples

The use of functional examples is to uniquely identify the purpose of the function. It is a good idea to let the students give some functional examples in a way they can also be used to test the function. This is in [HtDP] clarified with givens and expects. If you have the function and you input the given value, you want the returned value to be the expect. In [Coursera] the examples are already wrapped in check-expect statements. These check-expect statements should be written so that they fail at first. Then code is written to make the test pass.

When you first write the expectations of the function, you are already thinking about what the function should do, and what the results should be. This is also an important part of the test driven development framework. In a number of articles about this subject in the learning environment, it is called Test Driven Learning. There is a style which is called test-first and one that is called test-last.

According to [Janzen and Saiedian, 2006], it is valuable for students to learn how to construct programs in a test-driven manner, because initial evidence indicates that TDL can improve student comprehension of new concepts while improving their testing skills with no additional instruction time.

In addition, by learning to construct programs in a test-driven manner, students are expected to be more likely to develop their own code with a test-driven approach, likely resulting in improved software designs and quality.

Both [Sommerville, 2011] and [Langr, 2001] mentions beside better problem understanding four other positive effects of TfD/TDD. This better understanding is due to the fact that you've already written the test for it. This first (hopefully obvious) effect of TfD, is that the code ends up being testable [Langr, 2001]. In contrast, it is often extremely difficult, if not impossible, to write effective unit tests for code that has already been written without consideration for testing.

According to [Proulx, 2009] the professional programming community has realized that designing tests after the program has been written often leads to tailoring the tests to the code, rather than the original problem statement. Results from the research of [Janzen and Saiedian, 2008] indicate that a test-first approach can increase student testing and programmer performance, but that early programmers are very reluctant to adopt a test-first approach, even after having positive experiences using TDD.

According to [Edwards, 2003] TDD is attractive for use in education for many reasons. The student should be given the responsibility of demonstrating the correctness of his or her own code from the very first programming activities.

The instructors of upper level courses uniformly comment on students' better preparation [Proulx, 2009].

[WAC] uses the libraries Mocha[3] and Chai[4] to test the software. The students have to write their tests in the assert style of Mocha. The student can test his work before submitting his solution to the JavaScript–examiner to reduce the possibility of submitting code that will cause test performed by the JavaScript–examiner to fail.

2.4 Take inventory, to understand what are givens and what we do need to compute

The purpose of this step is to take the signature, and create a simple body, where only the return value is given. It doesn't compute anything, but gives the framework for the actual implementation of the function. In the JavaScript–examiner, this can also be checked against the signature. It can determine whether the function needs a return value or not. And it can check if it has the right return value. The body of the function is simplified, and can be filled with pseudo code. To check this can be a challenge, because of the diversity of statements. Another important thing is to distillate the return type. Again, this could generate useful feedback for the student.

[3] http://mochajs.org/
[4] http://chaijs.com/

2.5 Replace the body of the function with an expression that attempts to compute from the pieces in the template what the purpose statement promises

This is the actual implementation of the function. This should not be part of the recipe section in the submitted code, but be the actual submitted solution. The checking of this code in this section is the main function of the JavaScript–examiner.

2.6 Test the function on the examples that you worked out before

The submitted code should pass all the tests that are part of the solution. It should also pass the tests that are submitted by other students and have been evaluated as useful.

If there is a test that fails, an appropriate feedback should be returned. Then the student can adjust his code, which is known as refactoring, and submit his solution again.

The tests that are submitted by the student can also be used to test the submissions of other students. This way the library with available tests for a given exercise can be enriched. The tutor can supply standard tests. He can also keep track of the tests submitted by other students to look for duplicates. Also the tutor has to supply a standard solution to run the tests against, to check whether the test written by the students are correct. Tests that do not have the correct outcome against the standard solution, are not added to the test set.

The test that is part of the submission should have been created in an earlier step. Then this can be regarded as test-first development. However, if the tests are created within this step, after implementing the body, then this is test-last development. Because the solution is submitted as a whole, it will be very difficult to determine when the student has written his test.

Learning how to write tests is part of [WAC], but is not further discussed in this paper.

3 Conclusion

[HtDP] and Coursera] are examples of courses that use a design recipe for teaching how to learn to program in a structured manner, but they use it as a guide line in the Dr. Racket environment.

To promote the usage of the design recipe, the steps it contains should be part of the solution the student submits.

The JavaScript–examiner should have the recipe, commented out, as part of the submitted solution. Because the comment is not part of the Abstract Syntax Tree, it will have to be checked in an other way. If it can be included, this could be a requirement to pass the exercise. Both have to be analysed

further, because it can also lead to reluctance to use the JavaScript–examiner, but this can be subject of an other study.

It will require some effort of the tutor, to keep the list of tests free from duplicates, but as he manages to do this, it will be a great enhancement of the JavaScript–examiner.

At first, a student may feel that they are doing a lot of work, for relatively small function. But the goal should be learning the design recipe. The JavaScript–examiner is a good means to do this. A requirement should be that the JavaScript–examiner generates proper feedback,

This way, it can be a great aid in learning the recipe, and thus, learning the student to program better/make better functions.

The use of the recipe also pushes the student to use a TDD style.

It is also recommended to enhance the JavaScript–examiner to collect data out of the submitted solutions. This way future solutions can be compared and tested against already available data

Part of a following study could be how to determine when a test was written and so investigate the difference between test-first and test-last development.

Taken all this into account, it can be concluded that the use of a design recipe can be a very valuable extension of the JavaScript–examiner.

Bibliography

Bieniusa, A., Degen, M., Heidegger, P., Thiemann, P., Wehr, S., Gasbichler, M., Sperber, M., Crestani, M., Klaeren, H., and Knauel, E. (2008). Htdp and Dmda in the Battlefield: A Case Study in First-year Programming Instruction. In *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education*, FDPE '08, pages 1–12, New York, NY, USA. ACM.

Edwards, S. H. (2003). Rethinking Computer Science Education from a Test-first Perspective. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 148–155, New York, NY, USA. ACM.

Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. (2015). How to Design Programs.

Janzen, D. and Saiedian, H. (2008). Test-driven Learning in Early Programming Courses. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 532–536, New York, NY, USA. ACM.

Janzen, D. S. and Saiedian, H. (2006). Test-driven Learning: Intrinsic Integration of Testing into the CS/SE Curriculum. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '06, pages 254–258, New York, NY, USA. ACM.

Kiczales, G. (2015). Introduction to Systematic Program Design - Part 1 - The University of British Columbia.

Langr, J. (2001). Evolution of Test and Code Via Test-First Design.

Proulx, V. K. (2009). Test-driven Design for Introductory OO Programming. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, pages 138–142, New York, NY, USA. ACM.

Ramsey, N. (2015). On Teaching How to Design Programs (Abstract).

Sommerville, I. (2011). *Software engineering*. Pearson, Boston, 9th ed edition.