Beautiful JavaScript: How to guide students to create good and elegant code

Harrie Passier and Sylvia Stuurman and Harold Pootjes
Faculty of Management, Science and Technology, Department of Computer Science, Open Universiteit Nederland, Valkenburgerweg 177, 6419 AT Heerlen, The Netherlands

Programming is a complex task, which should be taught using authentic exercises, with supportive information and procedural information. Within the field of Computer Science, there are few examples of procedural information that guide students in how to proceed while solving a problem. We developed such guidelines for programming tasks in JavaScript, for students who have already learned to program using an object oriented language.

Teaching JavaScript in an academic setting has advantages and disadvantages. The disadvantages are that the language is interpreted so there is no compiler to check for type errors, and that the language allows many 'awful' constructs. The advantage is that, because of those disadvantages, programmers should consciously apply rules for 'good' programs, instead of being able to rely on the errors and warnings that a compiler will raise.

In this article, we show how we guide students to develop elegant code in JavaScript, by giving them a set of guidelines, and by advising a process of repeated refactoring until a program fulfills all requirements. To show that these guidelines work, we describe the development of a generic module for client-side form validation. The process followed and the resulting module both are valuable in an educational setting. As an example, it shows and explains precisely to students how such a module can be developed by following our guidelines, step by step.

Categories and Subject Descriptors: D.2.13 [**Reusable Software**]: Domain engineering—*Reusable libraries*; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Curriculum*

General Terms: Development of professional competencies in Computer Science, Programming guidelines

Additional Key Words and Phrases: Procedural guidelines, Programming guidelines, Design principles, Javascript, Web Application, Client-side, Form validation, Module pattern, Modularity

1. INTRODUCTION

Learning how to program according to general principles can be seen as a complex task. Learning complex tasks should be based on real-life authentic tasks [Merrill, 2002]. Students should be provided with support and guidance while solving such a task. That guidance should tell students how to recognize an acceptable solution and should provide guidance to the solution process: procedural information is required when one offers authentic tasks to enable complex learning [Kirschner et al., 2006]. This means that one should provide procedural information with programming tasks when one teaches students how to program. Procedural information specifies for learners how to perform the routine aspects of learning tasks, and preferable takes the form of direct, ste-by-step instruction [van Merriënboer and Kirschner, 2013].

Offering procedural guidance, in the form of a step by step approach to problem solving, is not very common in Computer Science education, as far as we know. We think that courses on programming could benefit from step by step approaches.

We developed a set of guidelines to create programs in JavaScript. The programming language Javascript has been developed within a short period of time, largely out of sight of academia. Javascript does not encourage encapsulation or structured programming, but strives to maximize flexibility, which may be a consequence of the fact that it was designed to allow non-programmers to extend web pages with logic [Richards et al., 2010]. As a result, Javascript has a number of awful and bad parts [Crockford, 2008]. When using JavaScript, it is therefore very easy to create unstructured programs that do not satisfy general design and programming principles; programs with, as a consequence, a low level of reusability, understandability or adaptability.

On the other hand, Javascript has many good parts too [Crockford, 2008] and it is possible to create elegant code in JavaScript, when adhering to certain principles.

Precisely the lack of language concepts such as modularity or information hiding makes Javascript a suitable language for learning how to program according to those principles. Students will have to force themselves to program in a clean way, according to programming principles, instead of being forced to do so by the language and/or the compiler.

Because the language does not enforce encapsulation and does not have easy solutions for private members or interfaces, students are forced to consciously apply those concepts to their programs, and as a result will become more aware of the value of the principles of, for instance, separation of concerns, information hiding, abstraction or modularity. As a consequence, students will get a more profound understanding of these principles.

By using our guidelines, students should be able to write elegant code.

As a first step in the validation of our guidelines, we have followed them to create a module for form validation. We show that a student may start with a program that is correct but does not comply with the software engineering design principles, and derive, step by step, elegant code that is easily extendible, and conforms to the design principles of software engineering.

In this paper, we describe the rules we give our students to recognize an acceptable solution, and we show the guidance for the process toward a solution. We will show how adhering to these guidelines leads to elegant code, by presenting a generic module for form validation as example.

Even though there are several existing libraries for form validation, form validation is a fruitful subject when teaching students to apply design principles as abstraction, modularity, extendability and information hiding to JavaScript code.

1.1 Contributions

We offer a set of guidelines to derive programming code that adheres to the design principles of software engineering. We show

how these guidelines lead to elegant code by applying them to an example problem: client-side form validation in JavaScript.

Some of the guidance is specific for form validation or for JavaScript, but most of the guidance is applicable to other programming languages.

The resulting module and the process followed is an excellent illustration of the fact how students could write elegant JavaScript when they would follow the rules we set out for them. What needs to be done, howeverm, is to validate whether the guidelines really help them in practice.

1.2 The structure of the paper

An environment for complex learning should have the following components [Van Merriënboer et al., 2002]:

Task description the task itself, which should be as authentic as possible. In this case, this is the task to create a module for client-side form validation. In section 2, we describe the general requirements of a form validation module to describe the task.

Supportive information may be related to the domain of the task, or describes how students may recognize that a solution is acceptable. In section 3, we describe both the specific rules to which a solution for a form validation module must adhere, and general design principles that must hold for any program.

Procedural information describes how students may proceed, which steps they should take while trying to solve the problem. In section 4 we introduce the procedural information, in the form of a set of step by step guidelines. We show guidelines that are specific for this domain, as well as guidelines that apply to any program.

Part-task practice items provide exercises that help students to reach a higher level of automation. We will not address these (smaller) exercises in this paper.

In section 5, we show that following our guidelines indeed leads to code that adheres to the design principles of section 3. In section 6, we discuss related work, and finally, in section 7, we discuss our own work, draw our conclusions and define future work.

2. TASK DESCRIPTION: REQUIREMENTS

2.1 Validation of input data

Form validation is the process of comparing data entered in a form against specifications for these data. Common validation procedures include checking for missing data (such as missing a social security number), invalid data (such as an impossible zip-code), and inconsistent data (such as a not existing combination of address and zip code) [Oliveira et al., 2005]. Some of these validation checks are obvious, such as a name - password combination. Other checks require more complex processing, such as checking the existence of a combination of address and zip code.

Validation of input data is needed both to keep the data stored on the server accurate and for security reasons, for instance, to prevent integer overflow and SQL injections [Pietraszek and Berghe, 2006]. The server thus must always validate any incoming data before storing them.

For usability reasons, however, it is preferable to validate entered data on the client as well. If not, a user would have to fill out a form, submit it, and wait for the response page that might tell the user that some of the entries are missing or are invalid. Experiments suggest that feedback should be given immediately after a user submits a form (and not after having entered a single field because users appear to get confused when that happens), and should be embedded in the form [Bargas-Avila et al., 2010]. Specifications for form entries should adhere to the robustness principle [Postel, 1981], which means that one should not be too strict on the format that the user should adhere to, and translate the entered data into a format that the server expects. It should be possible, for instance, to enter a zip code both with and without a space between the numbers and the letters, and it should be possible to enter the letters both capitalized or not.

The four dimensions of intrinsic information quality are authorization of the person who entered the data, timeliness of data entry, and correctness and completeness of the data entered [Ballou and Pazer, 1985]. Authorization can only be checked on the server, and the same applies for timeliness. For client-side form validation, therefore, the focus is on completeness and correctness of the data entered. Completeness means that all data needed is entered, i.e. there are no empty but required fields; correctness means that all these data is correct in relation to a certain specification. Correctness only needs to be checked if a field is complete (either has a value or is optional).

Authorization in the context of client-side form validation can be considered as a check on completeness (do the fields for the person's name and password contain data) as well as on correctness (are the person's name and password entered a valid combination).

Browsers that support form features of HTML5 already validate user input according to the type attribute (such as date). There are even some browsers that validate user input according to the value of the pattern attribute. In these browsers, validation using JavaScript is not really necessary, but it is still advisable, because the feedback browsers give is often confusing. Also, there still are browsers that do not have this functionality, and older versions of browsers, without this functionality, will be used for a long time. Furthermore, browsers are not able to validate related input fields such as street name, house number, city name, and zip code. Client-side form validation in JavaScript thus is still necessary, and will be necessary for the coming years.

2.2 Requirements for the solution

As we have seen, client-side form validation should focus on completeness and correctness. Completeness of an input field means that the field, if required, is not empty. Correctness of an input field means checking the following properties:

—Form data are received, by the script, as strings. Those strings should be parsed to check whether the input has the expected format (according to the robustness principle). For example, a (Dutch) zip code is formed by a sequence of four numbers, an optional space, and two letters.

—Is the data entered within a certain *range* defined by a minimal and maximal value? For example the age of a person should be between eighteen and hundred and twenty.

—Is the *combination* of data entered correctly? For example, the street name, house number, zip code and city name entered should be a valid combination.

Combinations of type and range will often occur, for example, an integer within a certain range or a string value out of an enumeration of strings. A variation of the third property is to require only a minimum or maximum value (for example, a person should be 18 years or older). Of course, there is always a default minimum and maximum: the minimum and maximum supported by JavaScript.

The last property of correctness (combination) is often validated on the server, because one needs to look up data in a database,

which will in general not be available at the client-side. This falls outside the scope of this task.

3. SUPPORTIVE INFORMATION

For each software product, there are general design principles and programming principles. For this domain, there are specific design principles for form validation. Of course, general knowledge on JavaScript and HTML also falls in this category, but for the purpose of this paper, we focus on the design principles.

3.1 General design principles

The Software Engineering Body of Knowledge [Bourque and Fairly, 2014] contains the following software design principles:

Abstraction Abstraction is 'a view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information' [Allen et al., 2009] and may consist of removing detail (to simplify and focus attention) and to generalize and identify the common core or essence [Kramer, 2007]. Abstraction can be used as a means to decrease complexity or to enhance useability.

Coupling and cohesion Coupling is the interdependence among modules, classes or functions (which should be minimized), and cohesion is the strength of association of the elements within a module, class or function (which should be maximized) [Allen et al., 2009].

Decomposition and modularization Software is divided into a number of smaller components with well-defined interfaces. This is usually accompanied by separation of concerns, placing different responsibilities in different components.

Encapsulation/information hiding Internal details of an abstraction are hidden, not available to external components.

Separation of interface and implementation This can be seen as a form of information hiding: components offer a public interface, and hide their implementation.

Sufficiency, completeness and primitiveness To be sufficient and complete, a software component captures the important characteristics of an abstraction and nothing more. Primitiveness means that simple is better than complicated.

Separation of concerns The notion of separation of concerns has been coined by Dijkstra [Dijkstra, 1982]. A concern in software architecture is an interest of one or more of the stakeholders of a system, but in the expression 'separation of concerns', a concern is an aspect of the system that is being designed. Separating concerns is a means of handling complexity.

3.2 Specific design principles for form validation

For form validation in a web application, there are some specific, generally accepted, design principles:

Server-side validation is mandatory All data must be validated on the server, because client-side validation can easily be bypassed and/or a user can have switched off the Javascript engine. The server is responsible for security, and for guarding the integrity of the stored data. Client-side form validation has a different purpose: usability, and user experience.

Enough is enough Only those validation functions should be performed that 1) do not need secure information from the server, and 2) support the user with filling in a form. The first restriction means that, for example, validation of authorization information should not be performed at the client-side. The second restriction means that the focus of client-side form validation should be on user experience: clear information about the completeness and correctness of the data entered, and about the way a form should be filled in.

Robustness principle The robustness principle [Postel, 1981] says 'Be conservative in what you do, be liberal in what you accept from others'. A form is an interface with two faces: one to the application software and one to the user. Both faces have different requirements. For example, a software function might need a parameter of type integer, where a user might think in terms of 1, 1.00 or even 'one'; the form should – preferably – allow the user to enter 1, 1.00 or 'one', and the validation program should translate this input into an integer.
An interface satisfies the robustness principle if 1) no rigid demands on the user input exist, 2) automatic translation to internal representations occurs in those cases where the data entered is of a type that is unsuitable for the processing software, 3) the requirements on the data to enter are clear for the user, and 4) clear feedback is presented in cases of invalid as well as valid data (for example by showing a green check sign near the input field). Feedback should be presented after pressing the submit button.

Guide the user A user should be guided by implicit as well as explicit form validation. Implicit form validation means guiding a user by using clear labels (specifying what should be entered), placeholders (presenting an example of what should be entered), fieldsets with legends (grouping related fields together), and, for example, radio buttons or drop-down lists in cases where one item, or a limited number of items, should be chosen from an enumeration (limiting the number of possible values to enter) [Bargas-Avila et al., 2010].
Explicit form validation means that a user should be immediately (after submitting the form) informed by a feedback message about the validity of data entered.

Aim for reusability Reusability means that software elements may serve for the construction of many different applications [Meyer, 1997]; reusability is an obvious principle in case of form validation, because forms often ask the same data, as for example name, address, zip code, city, and phone number. As a result, the same types of validation have often to be performed.

Aim for extensibility Because variations in input data exist, the programming code should be easily extendible (new validation functions can be easily added) as well as adaptable (existing functions can easily be changed).

3.3 Programming rules for JavaScript

We supply our students with several programming rules for JavaScript. Examples are:

—Declare constants and variables before anything else.

—Always use the keyword `var` to declare a variable and the keyword `const` to declare a constant.

—Use capitals for constants.

—Use object chaining if possible.

—Put related functions and variables into a module, using the module pattern [Stefanov, 2010], and compose a public interface with care.

These rules help students to structure their code.

4. PROCEDURAL INFORMATION

For the development of a form and the associated validation functions, we use a two layer architecture: 1) a GUI-layer, describing the static aspects of a form, specified in HTML5, and 2) a domain layer, describing the dynamic aspects of the form, specified in Javascript. Each of these layers have their own developing steps. A data model, which we have to specify first, is used in both these layers.

4.1 The model

Before the GUI-layer and the domain layer can be developed, one should determine which data we need from the user for this application. To specify these data, follow these steps:

4.1.1 *Specify the data that should be entered.* The purpose of this step is to determine the type for each data item. We distinguish between string and integer (positive or negative whole numbers).

—In many cases, valid input values are limited to an enumeration. If this is the case, specify this enumeration.

—In case of a string with requirements on structure, specify this structure expressed in the form a regular expression. Users are not supposed to enter numeric values according to the specific format they have in JavaScript. Instead, when users enter a decimal or another numeric value, the input is read as a string, and parsed against a regular expression. This allows us to adhere to the robustness principle. Not every format can be expressed in the form of a regular expression. In such cases, the validation on the server-side will be more severe than the validation on the client-side. For validation on the client-side, regular expressions suffice.

—In case of integers, determine minimum and/or maximum values if applicable.

—In all other cases there are no requirements on correctness, which means that the format is 'free-form' text, for example, a text field to write an opinion of a certain product.

4.1.2 *Specify user guidance*

—Specify user guidance for each data item, indicating how the form element should be filled in.

—Create an example value for each item.

—Create a label for the item.

—Specify whether a value has to be entered or whether this is optional.

4.2 The GUI layer

4.2.1 *Determine suitable* HTML5 *elements.* Based on the first step, determine for each data input a suitable HTML5 element, satisfying the principles of implicit form validation. Examples are the use of an HTML radio-button or drop-down list in case of one item or a limited number of items out of an enumeration, or the use of a HTML5 input element with type attribute 'date' in case of date selection.

—Place a `span` element right to each `input` field. This element will be used to display feedback, in cases of correct or incorrect data.

—Provide each `input` element with the user guidance that you have specified for the `title` attribute. The `title` attribute is used to guide the user in determining how to fill in the input element.

—Use the example value for the `placeholder` attribute.

—Use the name for the item for the `label` .

—Give each `input` element a unique `id` .

—Give the `input` element the attribute `required` unless it is optional.

—Finally, in cases of requirements on structure or of integer values that should be within a range, specify the corresponding values for the `pattern` attribute (a regular expression) or the `min` and/or `max` attribute.

Furthermore, group items together in fieldsets and take care of clear legends.

4.3 The domain layer

Although experienced developers may be able to design and implement code that will at once satisfy the principles, less experienced developers, as students often are, can reach this level of design and code only by a step by step approach in which refactoring is an important tool.

First, we give guidance on how to achieve a working application in this domain of form validation. Then we show the guidelines we give for every JavaScript application to refactor it into an application that adheres to the design principles described before.

4.3.1 *Step 1: Create a form validation application.* To create a form validation application, follow the following steps:

(1) Specify and implement checking for completeness. To determine whether the form is complete, create a function that tests whether each `input` element with the attribute `required` has a value. Checking involves: read the data from the HTML `input` element, determine whether there is an attribute `required` and a value, and return `false` when a value is required but absent; otherwise, return `true` .

(2) Specify and implement checks on correctness: depending on the specification of correctness of each required `input` field, create a function that tests whether the value conforms to these specifications. This involves: read the data from the HTML `input` element, perform validation on correctness, write a feedback message to the belonging `span` element, and return `true` or `false` . If there are no requirements specified on correctness, validating an `input` element returns `true` . If there is no value, this function should also return `true` : if the check of correctness is called, the check on completeness did tell that the field was either optional or contained a value.

(3) Create an event handler for the `submit` button. This event handler first checks for completeness and then for correctness of all required fields. For now, it prints to the console whether the form is valid or not. Here, one may also check on combinations of values (this is outside the scope of this paper).

(4) Check the code with a tool such as JSHint. [1]

(5) Create a test set and test the code.

4.3.2 *Step 2: Refactor the code.* Unlike the rules that guide a student in creating a form validation application (step 1), the guidelines for refactoring are general: these guidelines will be part of the procedural guidance in every JavaScript application.

These guidelines will sometimes have to be followed multiple times, depending on the results of the evaluation in the next step. Refactoring is driven by the following steps:

[1] http://www.jshint.com

(1) Remove duplicated code. If there are multiple instances of pieces of code, create helper functions to remove the duplication.
(2) Think of different extensions that might be needed in the future, and check where changes would have to be applied in the code. If such an extension needs changes in different locations, try to restructure the code such that the extension will become easier to apply.
(3) Check for cohesion within functions. Also, review the code with respect to sufficiency, completeness and primitiveness. If a function has more than one responsibility, split it in several functions.
(4) Review the code with respect to separation of concerns. Separate the code that interacts with the DOM (this can be seen as the controller) from the code that contains the program logic.
(5) Create modules for the controller and the program logic.
(6) Review the modules with respect to encapsulation/information hiding and separation of interface and implementation. Only the functions that are needed outside the module should be exported; nothing else.
(7) Review the code with respect to coupling. The model, for instance, should not need to call functions from the controller; only the other way around.
(8) Check the code with a style checker tool such as JSHint. [2]
(9) Test the code with the same test set as implemented in step 1.

As we will see in the next section, in particular removing duplication of code and enhancing changeability and abstraction are important tools in achieving the general design and programming principles described in section 3.1.

4.3.3 *Step 3: Evaluate, and document the design and implementation.* In this step, the design and implementation should be evaluated with respect to the general design and implementation principles as well as to the specific principles for validation. If the general and/or specific principles are not satisfied, the design and implementation must be changed by refactoring, along the guidelines to refactor the code of step 2. Also, the code should be completed with tests and documentation. Documentation can, of course, be written earlier, but in this step one must check whether the documentation is complete.

5. APPLYING THE PROCEDURAL INFORMATION, FIRST ATTEMPT

In this section we show how the procedural information, described in section 4, leads to elegant code, using an example problem. For reasons of space, we show a small problem: a form consisting of two input elements. Both fields are part of a form for applying for a car insurance. The first field is a zip code field; the second field contains the number of years the applicant has driven without an accident.

5.1 The model

5.1.1 *Specify the data that should be entered.* For the zip code, the type of data is `string` . Although, each zip code is actually one out of an enumeration, applying an enumeration is infeasible in practice because of the huge number of zip codes. Instead, we specify that each zip code has to satisfy a structure constraint: a sequence consisting of four digits, an optional space, and two letters. To be precise, in regular expression notation, the specification is:
$[1-9]\backslash d\{3\}\backslash s?[a-zA-Z]\{2\}$

For the `input` field representing the number of years without accident, the type of data is `integer` . Assuming a minimum value of `0` and a maximum value of `70` years, we could specify the type of this field as an enumeration. Again, we consider this as impractical and specify this field as type `integer` with a minimum and maximum value.

5.1.2 *Specify user guidance.* As user guidance for the zip code, we specify: 'The correct format for the zip code is: four numbers between 0 and 9, followed by a space, followed by two characters (either uppercase or lowercase). Between the numbers and the characters, you may enter a space. Two examples of the correct format for the zip code are 1234 AB and 1234AB'. The label for the item will be 'Zip code'.

As user guidance for the `input` field representing the number of years without accident, we specify: 'The number of years without an accident is a whole number between 0 and 90'. An example is `17` , and the name of the item will be 'Number of years without accident'.

Both items are required.

5.2 The GUI

5.2.1 *Determine suitable* HTML5 *elements.* The zip code field is specified as an input field of type `text` . The placeholder attribute shows a correct example.

The field representing the number of years without an accident is specified as an input field of type `number` . The placeholder attribute shows a correct example.

Both items receive the attribute `required` .

The HTML-code:

```
<label class="heading">Zip code</label>
<input id="zipcode"
       class="input"
       type="text"
       name="zipcode"
       pattern="^[1-9]\d{3}\s?[a-zA-Z]{2}$"
       placeholder="1234 AB"
       title="The correct format for the zip code is:
            four numbers
              between 0 and 9, followed by a space,
                   followed by two
              characters (either uppercase or lowercase).
              Between the numbers and the characters, you
                    may enter
              a space."
       required>
<span class="feedback"></span>

<label class="heading">
  Number of years without accident
</label>
<input id="yearswithout"
       class="input"
       type="number"
       name="yearsWithoutAccident"
       min="0"
```

[2] http://www.jshint.com

```
      max="70"
      placeholder="17"
      title="The number of years without an accident is
            a whole number between 0 and 70."
      required>
<span class="feedback"></span>
```

Notice that each `input` element contains all information needed for validation, i.e. the value to validate and the specifications of completeness and of correctness, represented by the values of the attributes `pattern` , `min` and `max` . The responsibility of the script is to validate whether input of the user satisfies the restrictions specified in HTML; the responsibility of HTML is to specify which kind of input is asked for.

5.3 The JavaScript code

In this section, we follow the guidelines presented in subsection 4.3, to show how students will achieve an elegant, flexible form validation module.

5.3.1 *Create a form validation application.* To illustrate how the refactoring guidelines work, we start with a 'naive' implementation, with a validating function for each of the form items.

First, we specify a function to check for completeness. To do this, we define a function to check the completeness of a single item, that writes a message in its feedback field if a value is missing. We use the JavaScript library jQuery. Here, we show how we declare all constants at the top of the code; in the next pieces of code, we omit the constants we already showed, for reasons of space. Likewise, we do not show tests and documentation.

```
1  const CLASS_FEEDBACK = ".feedback",
2        INVALID        = "invalid",
3        MISSING        = "You probably forgot this entry",
4        REQUIRED       = "required",
5        SUBMIT         = "#submit",
6        VALID          = "valid",;
7
8  function isComplete(el) {
9    var fbField = el.next(CLASS_FEEDBACK),
10   complete = !el.prop(REQUIRED) || el.val();
11   if (!complete) {
12     fbField.removeClass(VALID).addClass(INVALID).html(
           MISSING);
13   }
14   return complete;
15 }
```

Notice that the expression in line 10 describes exactly the definition of completeness, i.e. if a field is required, then there should be a value entered. To test on correctness of the zip code we write a function that returns `true` if there is a value that conforms to the regular expression specified by the pattern, or if there is no value. The function returns `false` when the value does not conform to the specification. We only show the constants that we did not define before.

```
1  const OK     = "&#10003;",
2        PATTERN = "pattern",
3        TITLE   = "title",
4        ZIPCODE = "#zipcode";
5  function zipCodeIsCorrect() {
6    var isCorrect = true,
7        element  = $(ZIPCODE),
8        fbField  = element.next(CLASS_FEEDBACK),
9        value    = element.val(),
10       pattern  = element.attr(PATTERN),
11       regex    = new RegExp(pattern),
12       title    = element.attr(TITLE);
13   if (value) {
14     isCorrect = regex.test(value);
15     if (isCorrect) {
16       fbField.removeClass(INVALID).addClass(VALID).html(
             OK);
17     } else {
18       fbField.removeClass(VALID).addClass(INVALID).html(
             title);
19     }
20   }
21   return isCorrect;
22 }
```

Because of the specification of the HTML input elements, the code of other event handlers for validating correctness against a pattern is almost the same. Most of the code could be copied and only the `id` would have to be changed (which means we already see a reason to refactor later).

The implementation of the event handler validating the number of years without an accident could be as follows:

```
1  const MAX         = "max",
2        MIN         = "min",
3        YEARSWITHOUT = "#yearsWithoutAccident";
4  function yearsWithoutAccidentIsCorrect() {
5    var isCorrect = true,
6        element  = $(YEARSWITHOUT),
7        fbField  = element.next(CLASS_FEEDBACK),
8        value    = parseInt(element.val()),
9        min      = element.attr(MIN) || Number.MIN_VALUE,
10       max      = element.attr(MAX) || Number.MAX_VALUE,
11   title = element.attr(TITLE);
12   if (value) {
13     isCorrect = value >= min && value <= max;
14     if (isCorrect) {
15       fbField.removeClass(INVALID).addClass(VALID).html(
             OK);
16     } else {
17       fbField.removeClass(VALID).addClass(INVALID).html(
             title);
18     }
19   }
20   return isCorrect;
21 }
```

At last, we create an event handler for the submit button, that prevents the browser from submitting the form, and checks whether the input is complete and correct. The application is now complete (apart from the fact that we do not include the code for the Ajax call to the server in the case when the input is complete and correct).

```
1  const CLICK = "click";
2  $(document).ready(function () {
3    $(SUBMIT).on(CLICK, isSubmittable);
4  });
5
6  function isSubmittable(event) {
```

```
7   var zipcodeComplete = isComplete($(ZIPCODE)),
8       yearsWithoutAccidentComplete = isComplete($(
            YEARSWITHOUT)),
9       complete = zipcodeComplete &&
            yearsWithoutAccidentComplete,
10      zipcodeCorrect = zipCodeIsCorrect(),
11      yearsWithoutAccidentCorrect =
            yearsWithoutAccidentIsCorrect(),
12      correct = zipcodeCorrect &&
            yearsWithoutAccidentCorrect;
13  event.preventDefault();
14  return complete && correct;
15 }
```

5.4 Refactor the code

When reviewing the Java Script code from the previous subsection, it is clear that a number of general as well as specific principles are not sufficiently met. We will follow the guidelines to improve our code.

5.4.1 *Remove duplicated code.* The most obvious occurence of duplication is the handling of the feedback field. The guideline tells us to create a helper function. This function should receive information about the class (valid or invalid), and about the message that should be shown. To transfer this information, we create a constructor `Message` for message objects (line 1), and a function `writeMessage` that receives a message object and an element to write to (line 5). We also write a function `handleMessage` that creates the right message, based on the value of `correct` (line 13).

```
1  function Message (valid, feedback) {
2    this.valid = valid;
3    this.feedback = feedback;
4  }
5  function writeMessage(field, message) {
6    if (message.valid) {
7      field.removeClass(INVALID).addClass(VALID).html(
           message.feedback);
8    }
9    else {
10     field.removeClass(VALID).addClass(INVALID).html(
           message.feedback);
11   }
12 }
13 function handleMessage(correct, field, errorString) {
14   var message = new Message(true, OK);
15   if (!correct) {
16     message.valid = false;
17     message.feedback = errorString;
18   }
19   writeMessage(field, message);
20 }
21 function isComplete(el) {
22   var fbField = el.next(CLASS_FEEDBACK),
23       complete = !el.prop(REQUIRED) || el.val(),
24   if (!complete) {
25     handleMessage(complete, fbField, MISSING);
26   }
27   return complete;
28 }
29 function zipCodeIsCorrect() {
30   var isCorrect = true,
31       element  = $(ZIPCODE),
32       fbField  = element.next(CLASS_FEEDBACK),
33       value    = element.val(),
34       pattern  = element.attr(PATTERN),
35       regex    = new RegExp(pattern),
36       title    = element.attr(TITLE);
37   if (value) {
38     isCorrect = regex.test(value);
39     handleMessage(isCorrect, fbField, title);
40   }
41   return isCorrect;
42 }
43 function yearsWithoutAccidentIsCorrect() {
44   var isCorrect = true,
45       element  = $(YEARSWITHOUT),
46       fbField  = element.next(CLASS_FEEDBACK),
47       value    = parseInt(element.val()),
48       min      = element.attr(MIN) || Number.MIN_VALUE,
49       max      = element.attr(MAX) || Number.MAX_VALUE,
50       title    = element.attr(TITLE);
51   if (value) {
52     isCorrect = value >= min && value <= max;
53     handleMessage(isCorrect, fbField, title);
54   }
55   return isCorrect;
56 }
```

In lines 25 and 39, we now see a call to the function `handleMessage` instead of two similar pieces of code that change the feedback field directly.

5.4.2 *Review changes in case of extensions.* The first possible extension that springs to mind, is to add another item to the form. In the application as it has been structured at this moment, a separate function should be written for each item. It would be nice if the application could be used for every form; not just for this particular one.

An input value is either a number (for which a range might be set) or a text which might have to obey certain rules. This means that there will be two general validating functions:

```
1  function isValidAgainstRegex(element) {
2    var isCorrect = true,
3        fbField = element.next(CLASS_FEEDBACK),
4        value   = element.val(),
5        pattern = element.attr(PATTERN),
6        regex   = new RegExp(pattern),
7        title   = element.attr(TITLE);
8    if (value) {
9      isCorrect = regex.test(value);
10     handleMessage(isCorrect, fbField, title);
11     }
12   return isCorrect;
13 }
14
15 function isValidNumber(element) {
16   var isCorrect = true,
17       fbField  = element.next(CLASS_FEEDBACK),
18       value    = parseInt(element.val()),
19       min      = element.attr(MIN) || Number.MIN_VALUE,
20       max      = element.attr(MAX) || Number.MAX_VALUE,
21       title    = element.attr(TITLE);
22   if (value) {
```

```
    isCorrect = isValid = value >= min && value <= max;
    handleMessage(isCorrect, fbField, title);
  }
  return isCorrect;
}
```

We would like to rewrite `isSubmittable` in such a way that we can call one and the same function for each input. Note that, here, `isComplete` and `isCorrect` have side-effects to show feedback, and that the side-effect of `isCorrect` is only needed when the input is complete.

```
const INPUTS = "input:not(#sumbit)";

function isSubmittable(event) {
  var complete = true,
      correct = true;
  event.preventDefault();
  $(INPUTS).each(function(index, element) {
    if (!isComplete($(element))) {
      complete = false;
    }
    else {
      if (!isCorrect($(element))) {
        correct = false;
      }
    }
  });
  return complete && correct;
}
```

This means that we have to implement a function `isCorrect`, which chooses, depending on the `type` of an item, which correctness checking function to use.

```
function isCorrect (element) {
  var res = true;
  switch (element.attr(TYPE)) {
    case RANGE :
    case NUMBER : {
      res = isValidNumber(element);
      break;
    }
    default : {
      if (element.attr(PATTERN)) {
        res = isValidAgainstRegex(element);
      }
      break;
    }
  }
  return res;
}
```

5.4.3 *Check for cohesion, sufficiency, completeness and primitiveness.* With respect to cohesion: `isValidAgainstRegex` and `isValidNumber` write feedback as a side-effect. This has a negative effect on primitiveness: those functions could be given less responsibilities. It is not clear where we should write the feedback, so we postpone that decision until a later step in the refactoring.

For now, the application seems to be sufficient and complete with respect to various form items. One exception is that we assumed that every item with type `range` involves integers, meaning that the attribute `step` has the default value of 1. That attribute may also have a value of, for instance, 0.5 or 0.1. In that case, we would need another validating function, and `isCorrect` would have to be adapted. We will leave this for now, but remark that the set of guidelines help us in observing such an extension, which would make the validation module more generally usable.

5.4.4 *Review with respect to separation of concerns.* The separation between the HTML and the JavaScript is clean: in the HTML, the type of each input item is specified, and in some cases a minimum, maximum, or a pattern is specified as well. Also, the HTML specifies whether an item should be given a value. In the script, a validation function is chosen based on the specifications, and the required items are checked on the presence of a value.

Within the JavaScript, however, there is no separation of concerns. The first action could be to divide the code into two files. One file, `controller.js`, contains everything that addresses the DOM (changing the DOM, binding event handlers), while the other file, `validator.js`, contains the validating functionality.

This guideline also solves the problem we faced: writing the feedback to the DOM should be in `controller.js`. Because `isSubmittable` is an event handler for the submit button, this is the logical place to write the feedback. Instead of having the validating functions call `writeMessage`, the validating functions now return a boolean: the validating functions now only do what their name suggest: they validate.

The functions in `validate.js` now have no side-effects; the only function with a side-effect is `isSubmittable` in `controller.js`, and the only purpose of that function is to have side-effects: to give feedback to the user, and, if the form is complete and correct, to send the form to the server. We might decide for a different name for this function, to show that it has side-effects, but for now, we leave the name as it is. As a bonus, we now do not need the constructor for `Message` objects anymore.

The file `validator.js` is as follows:

```
const MAX     = "max",
      MIN     = "min",
      NUMBER  = "number",
      PATTERN = "pattern",
      RANGE   = "range",
      REQUIRED = "required",
      TYPE    = "type";

function isComplete(el) {
  return complete = !el.prop(REQUIRED) || el.val();
}

function isCorrect (element) {
  var correct = true;
  switch (element.attr(TYPE)) {
    case RANGE :
    case NUMBER : {
      correct = isValidNumber(element);
      break;
    }
    default : {
      if (element.attr(PATTERN)) {
        correct = isValidAgainstRegex(element);
      }
      break;
    }
  }
  return correct;
```

```
}

function isValidAgainstRegex(element) {
  var isValid = true,
      value   = element.val(),
      pattern = element.attr(PATTERN),
      regex   = new RegExp(pattern);
  if (value) {
    isValid = regex.test(value);
  }
  return isValid;
}

function isValidNumber(element) {
  var isValid = true,
      value    = parseInt(element.val()),
      min      = element.attr(MIN) || Number.MIN_VALUE,
      max      = element.attr(MAX) || Number.MAX_VALUE;
  if (value) {
    isValid= value >= min && value <= max;
  }
  return isValid;
}
```

If, in the future, more validating functions are required, the only function that will require a change is the function `isCorrect` . An alternative is to specify the kind of validation that is required in HTML.

The file `controller.js` is as follows:

```
const CLASS_FEEDBACK = ".feedback",
      CLICK          = "click",
      INPUTS         = "input:not(#sumbit)",
      INVALID        = "invalid",
      MISSING        = "You probably forgot this entry",
      OK             = "&#10003;",
      SUBMIT         = "#submit",
      TITLE          = "title",
      VALID          = "valid";

$(document).ready(function () {
  $(SUBMIT).on(CLICK, isSubmittable);
});

function isSubmittable(event) {
  event.preventDefault();
  $(INPUTS).each(function(index, element) {
    var el = $(element),
      feedbackEl = el.next(CLASS_FEEDBACK);
    if (!isComplete(el)) {
      writeMessage(false, feedbackEl, MISSING);
    }
    else {
      if (isCorrect(el)) {
        writeMessage(true, feedbackEl, OK);
        // send form to server
      }
      else {
        writeMessage(false, feedbackEl, el.attr(TITLE));
      }
    }
  });
}

function writeMessage(valid, field, message) {
  if (valid) {
    field.removeClass(INVALID).addClass(VALID).html(
        message);
  }
  else {
    field.removeClass(VALID).addClass(INVALID).html(
        message);
  }
}
```

5.4.5 *Create modules.* Now, it is easy to create modules. The only functions that the controller needs of the validator are `isComplete` and `isCorrect` . The public API thus only consists of those functions.

```
var validator = (function (){
  // private
  const MAX     = "max",
        MIN     = "min",
        NUMBER  = "number",
        PATTERN = "pattern",
        RANGE   = "range",
        REQUIRED = "required",
        TYPE    = "type";

   var isComplete = function(el) {
    return complete = !el.prop(REQUIRED) || el.val();
  },

   isCorrect = function(element) {
    var correct = true;
    switch (element.attr(TYPE)) {
      case RANGE :
      case NUMBER : {
        correct = isValidNumber(element);
        break;
      }
      default : {
        if (element.attr(PATTERN)) {
          correct = isValidAgainstRegex(element);
        }
        break;
      }
    }
    return correct;
  },

   isValidAgainstRegex = function(element) {
    var isValid = true,
        value   = element.val(),
        pattern = element.attr(PATTERN),
        regex   = new RegExp(pattern);
    if (value) {
      isValid = regex.test(value);
    }
    return isValid;
  },

   isValidNumber = function(element) {
    var isValid = true,
```

```
      value    = parseInt(element.val()),
      min      = element.attr(MIN) || Number.MIN_VALUE,
      max      = element.attr(MAX) || Number.MAX_VALUE;
  if (value) {
    isValid= value >= min && value <= max;
  }
  return isValid;
 }
 // public API
 return {
  isComplete: isComplete,
  isCorrect: isCorrect
 }
}());
```

The controller may also be changed into a module, and does not have to reveal a public API.

```
var controller = (function (){
 // private
 const CLASS_FEEDBACK = ".feedback",
      CLICK         = "click",
      INPUTS        = "input:not(#sumbit)",
      INVALID       = "invalid",
      MISSING       = "You probably forgot this entry",
      OK            = "&#10003;",
      SUBMIT        = "#submit",
      TITLE         = "title",
      VALID         = "valid";

 var isSubmittable = function(event) {
  event.preventDefault();
  $(INPUTS).each(function(index, element) {
   var el = $(element),
    feedbackEl = el.next(CLASS_FEEDBACK);
   if (!validator.isComplete(el)) {
    writeMessage(false, feedbackEl, MISSING);
   }
   else {
    if (validator.isCorrect(el)) {
     writeMessage(true, feedbackEl, OK);
     // send form to server
    }
    else {
     writeMessage(false, feedbackEl, el.attr(TITLE))
          ;
    }
   }
  });
 },
 writeMessage = function(valid, field, message) {
  if (valid) {
   field.removeClass(INVALID).addClass(VALID).html(
        message);
  }
  else {
   field.removeClass(VALID).addClass(INVALID).html(
        message);
  }
 }

 $(document).ready(function () {
  $(SUBMIT).on(CLICK, isSubmittable);
 });
}());
```

5.4.6 *Review information hiding and separation of interface and implementation.* By encapsulating the code into modules, and by providing the smallest possible public API, our code already conforms to these rules.

5.4.7 *Review the code with respect to coupling.* In this case, the controller calls two functions of the public API of the validator; the validator does not need anything of the controller.

5.5 Evaluate and document

As a last step, we now review our code with respect to the design rules of subsection 3.1.

Abstraction The validator exports two abstract functions, and the controller does not refer to individual form items. The level of abstraction is high.

Coupling and cohesion The level of coupling is low: the controller uses two functions of the public API of the validator, and that is all coupling there is between the JavaScript files. There is coupling between the HTML and the script: the script relies on the `type`, `min`, `max`, `required`, and `pattern` attributes in the HTML. However, this is all within the HTML standard. The level of cohesion of the validator is high.

Decomposition and modularization The level of decomposition and modularization is high.

Encapsulation/information hiding The level of encapsulation and information hiding is high, because the public API of the validator contains only two functions. The functions of the controller cannot be reached from outside because the code is encapsulated in a module.

Separation of interface and implementation There is a clear separation, in the validator, between the interface (the public API) and the implementation.

Sufficiency, completeness and primitiveness The level of primitiveness is high, because each function now carries one responsibility. The only exception is the function `isSubmittable`, in which feedback is written as a side-effect. We think that is allowable, because checking whether the form may be submitted to the server and giving feedback if that is not the case is so closely related. The code is sufficient for form application. With respect to completeness, the code should be extended for ranges with steps other than 1.

Separation of concerns The level of separation of concerns is high.

Specific design principles From the specific design principles for form validation, the first four (i.e. server-side validation is obliged, enough is enough, robustness, and guide the user) are met or could be easily met. The last one, aim for reusability, is met also: this code is applicable to all forms (with the proposed extension with respect to steps other than 1).

6. RELATED WORK

Form validation using a specification declared in the HTML is not new. Powerforms [Brabrand et al., 2000], for instance, specifies additions to HTML that are parsed by JavaScript. There are more libraries for client-side form validation, such as the jQuery Validation plugin. Here, form validation is reduced to specifying which

fields have to be validated against which rules and, if a rule is broken, which feedback message should be presented. There are about fifteen different validation functions to choose from (for instance, email, creditcard or url). Our feedback library, while much barer, is simpler with respect to validating functions: all information is declared in the HTML, using standard attributes. However, our focus is not on the library itself. We have tried to show that the guidelines we give our students lead to elegant code, even if the first attempt is not really elegant.

One example of procedural guidelines on programming are detailed guidelines for functional programming [Felleisen, 2001]. We use similar guidelines for creating functions; we have not shown these guidelines here. The guidelines we present here are meant for event driven programs in JavaScript with a GUI in HTML.

Another example of procedural guidelines for programming is the STREAM framework [Caspersen and Kolling, 2009], a step by step approach intended for novices learning OO-programming. The approach is based on stepwise improvement consisting of extension (extending the specification to cover more use cases), refinement (refining abstract code to executable code), and restructuring (improving non-functional aspects of a solution without altering its observable behavior). The arguments for the necessity of procedural guidelines for students that the authors give are in line with our own observations. Our approach is geared toward more advanced programmers. Our guidelines are more general applicable, whereas the STREAM guidelines are specific for OO programs. Our guidelines are less specific, but we offer students a means to check their code with the principles that the code should adhere to when good enough. Students who did learn to program using the STREAM method, could learn advanced programming using our guidelines.

As far as we know, there are no other studies on the role of procedural information for the complex task of programming web applications and event driven programming.

7. DISCUSSION, CONCLUSION AND FUTURE WORK

Procedural information that help students solve complex problems is not available for many complex tasks in the domain of Computer Science. We developed such information for the task of programming in JavaScript. Programming in general is a complex task, and programming in JavaScript even more so, because there is no compiler to detect some types of mistakes, and the language allows one to write code that is hardly maintainable and difficult to analyse.

To validate the guidelines that we developed, we used them in an example task. We have described how our guidelines may help students to derive maintainable code that adheres to the software engineering principles. We have shown that sometimes a refactoring step may warn the student that there is a problem with the code, but does not give guidance in how to solve the problem. In such a case, one of the next steps may give more guidance to the solution.

We think that we should create step by step guidelines for the many complex tasks that we prepare our students for. This set of guidelines is one example. In our opinion, these guidelines are not simple-to-follow recipes that guarantee success. Instead, these guidelines help students to divide problems into smaller ones, to stimulate an attitude of first thinking, then doing.

The work that we have done not only shows that our guidelines might be helpful; we have also created an example case to explain our students how to use these guidelines.

What needs to be done, however, is to monitor whether these guidelines really help students. It is clear that the guidelines *may guide* students toward a good enough solution, but we will have to check whether they really *do so* in practice. We are currently working on a setup that will allow us to validate the working of these guidelines. In a situation of distance learning, we might use the Think aloud method in a session in our electronic learning environment, which allows us to record what the students does and says.

In the far future, we would like to work on tools to check automatically if a solution is good enough. Our guidelines could be used to provide meaningful feedback when a solution is not yet acceptable.

REFERENCES

Allen, J., Barnum, S., Ellison, R., McGraw, G., and Mead, N. (2009). *Software Security Engineering*. Addison-Wesley Professional.

Ballou, D. and Pazer, H. (1985). Modeling data and process quality in multi-input, multi-output information system. *Management Science*, 31(2):150162.

Bargas-Avila, J., Brenzikofer, O., Roth, S., Tuch, A., Orsini, S., and Opwis, K. (2010). Simple but crucial user interfaces in the world wide web: introducing 20 guidelines for usable web form design. *User Interfaces*, pages 1–10.

Bourque, P. and Fairly, R. E., editors (2014). *Guide to the Software Engineering Body of Knowledge version 3*. IEEE Computer Society.

Brabrand, C., Møller, A., Ricky, M., and Schwartzbach, M. I. (2000). Powerforms: Declarative client-side form field validation. *World Wide Web*, 3(4):205–214.

Caspersen, M. E. and Kolling, M. (2009). Stream: A first programming process. *ACM Transactions on Computing Education (TOCE)*, 9(1).

Crockford, D. (2008). *JavaScript: The Good Parts*. O'Reilly/Yahoo! Inc. First Edition.

Dijkstra, E. W. (1982). On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer.

Felleisen, M. (2001). *How to Design Programs: an introduction to programming and computing*. MIT Press.

Kirschner, P. A., Sweller, J., and Clark, R. E. (2006). Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational psychologist*, 41(2):75–86.

Kramer, J. (2007). Is abstraction the key to computing? *Communications of the ACM*, 50(4):36–42.

Merrill, M. D. (2002). First principles of instruction. *Educational technology research and development*, 50(3):43–59.

Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice Hall, Upper Saddle River, New Jersey, USA, second edition.

Oliveira, P., Rodrigues, F., and Henriques, P. R. (2005). A formal definition of data quality problems. In *International Conference on Information Quality ICIQ*, pages 14–24, Cambridge, Massachusstes, USA. MIT Information Quality (MITIQ).

Pietraszek, T. and Berghe, C. V. (2006). Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection*, volume 3858 of *Lecture Notes in Computer Science*, pages 124–145. Springer.

Postel, J. (1981). Request for comments 793-transmission control protocol.

Richards, G., Lebresne, S., Burg, B., and Vitek, J. (2010). An analysis of the dynamic behavior of javascript programs. *ACM Sigplan Notices*, 45(6):1–12.

Stefanov, S. (2010). *JavaScript patterns*. O'Reilly Media, Sebastopol, California, USA.

Van Merriënboer, J. J., Clark, R. E., and De Croock, M. B. (2002). Blueprints for complex learning: The 4c/id-model. *Educational Technology Research and Development*, 50(2):39–61.

van Merriënboer, J. J. and Kirschner, P. A. (2013). *Ten Steps to Complex Learning, a systematic appraoch to four-component instructional design*. Taylor & Francis, New York, NY, USA, second edition.