AutoRSpec

Dan Shreeve

2017-04-25

Signed declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations which are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: Daniel Demaine Shreeve

Signature: .....

Date: 03/05/2017

Abstract

Software Testing benefits the development and maintenance of a system by increasing quality and reliability. Software Testing also contributes around half of the cost of producing a system. Automating part or whole of this process reduces the cost whilst maintaining the benefits. The aim of this project is to produce a system that automatically generates RSpec test cases for model validation in Ruby on Rails. The tests will be generated from a formal database specification that the user has defined using the system. A file can be generated and inserted into the user's application and run as though they had been written by the user. The time taken to insert the necessary information should be less than the time taken to write the tests manually, otherwise the user will not benefit.

Acknowledgements

First and foremost I would like to thank my supervisor **Dr. Gordon Fraser** for accepting my proposed project and providing impeccable advice and guidance throughout the process.

Finally I would like to thank my parents, **Linda Shreeve** and **Paul Shreeve**, and my grandparents **Elsie Marsden** and **Ray Marsden** for giving me the oppurtunity to go to university and supporting me throughout.

Contents

1 Chapter 1: Introduction

How severe can the consequences be from an error in a piece of software? In 1983 a bug in a piece of software almost started World War Three.

During the Cold War, tensions between the US and Soviet Russia were extremely high. A Soviet early warning system had detected the launch of five ballistic missiles from the US. The only reason that Soviet Russia did not retaliate, thereby starting World War Three, was the fact that Lt Col Stanislav Petrov had a "...funny feeling in my gut"[1] and concluded that if the US was launching a full scale attack they would launch more than five missiles. The error in the system was discovered to be a bug in part of the software that distinguished false missiles from satellites picking up the reflection of sunlight from the top of clouds.[1] If the bug in the code had falslely detected more missiles the world could be a very different place today.

Software testing increases the relability of a system and quality, by detecting errors and bugs which can be fixed, quality can aslo be further improved by tests proving that the system meets its design requirements. However, software testing is very costly, accounting for half the time put into development of a system and half of total expenditure.[2] By automating part or all of the software testing process, testing costs can be reduced, whilst maintaining the benefits.

The motivation for this project is to bring the benefits of automated testing to Ruby on Rails. Ruby on Rails is a web application framework that allows developers to create fully functioning applications in a short of space of time. The reduction in time spent implementing increases the proportion of time spent testing. Thefore Ruby on Rails would benefit greatly from automated testing. This would further enhance the time saving feature of Ruby on Rails.

This project aims to create a system that Ruby on Rails developers can use to reduce the development time of their projects by automatically generating all or part of the tests they require. The tests generated should not reduce the reliability and quality that manually written tests create.

Chapter 2: Literature Review and Research gives detailed background information on software testing and how it can be automated. It then follows

how testing can be carried out in Ruby on Rails applications with a couple examples of existing teset automation in Ruby on Rails. Finally the chapter discusses the tools that will be used to construct this project.

Chapter 3: Requirements and Analysis discusss how the motivation is refined into a more spefici aim and gives the objectives and requirements that the project should have to achieve this aim.

Chapter 4: Design outlines the design of the project. Discussing and demonstrating vaarious aspects such as the database and value generation.

Chapter 5: Implementation and Testing shows how the design was implemented in detail broken down into sections on setup, database, value generation, test case templates and the overall test suite layout. It then shows how the project was tested and how the requirements from chapter 3 are met.

Chapter 6: Results and Discussion evaluates the project and discusses how effective the final implentation is against its aim.

Chapter 7: Conclusions discusses various ways in which the system could be improved and its potentail to be expanded. Then is wrapped up with a conclusion on the project as a whole.

2 Chapter 2: Literature Review and Research

2.1 Testing and Automation

The European Space Agency spent ten years and $7 billion designing and constructing the Ariane 5, a rocket that can launch multiple satellites into orbit from a single launch. Thirty nine seconds into its maiden voyage it exploded, destroying the Ariane 5 and its cargo of four uninsured, extremeley expensive scientific satellites. The explosion was caused by its own self-destruct sequence which was triggered automatically as the boosters were being torn away by aerodynamic forces. These extreme forces were caused by the rocket trying to recorrect its course in response to flight data provided by the guidance system. The guidance system, along with its backup, had crashed and shutdown. The flight data provided, that caused the rocket to readjust its course, was actually a diagnostic error message.

The cause of the shutdown was the guidance system trying to convert the sideways velocity of the rocket from 64-bit to 16-bit. To make matters worse, the programmers were aware that it could overflow but assumed that the particular causal variable would never be large enough, as it was used to prepare for launch and not in flight. However it was decided the system should run into the first forty seconds of flight, in case of a brief hold in the launch countdown, to make restarting the system easier. A known flaw in a system, that could have been handled, resulted in a chain of events that led to the catastrophic explosion of Ariane 5.[3].

Software disasters can be caused by poor testing practices[4], if the correct testing procedures and practices had been in place for both Ariane 5 and the Soviet Guidance System example from the Introduction these situations could have been avoided. Software testing is therefore extremely important and should be included in the development of all software applications. What is software testing and how does it help avoid these situations ?

Software testing is an investigation into a piece of software that provides information during development and maintenance. A process or series of processes are carried out that are designed to make sure computer code does what it designed to do and is absent of unintended behaviour[2]. The information retrieved from the proccesses can be used to track the progress

during development against acceptance criteria and detect and locate errors and bugs. Errors and bugs detected within the code of the are immediatley known and can be handled, providing a smoother and more consistent development and maintenance flow. Software testing provides a more reliable and higher quality product when used as part of the development process due to these benefits.

Testing however can not guarantee that a program or peice of code is without errors, therefor completing testing is impossible. This is why the design of tests is vital to the integrity of the testing, making the tests as complete as possible. Given the constraints on time and cost, effective testing is simply "What subset of all possible test cases has the highest probability of detecting the most errors?"[2]. Tests are designed using information about the program along with its intended behaviour. In a given environment, with proper determined input, there is an expected behaviour/output. If the code under test does not display the desired outcome it is said to have failed the test.

A 'test case' will test a very specific behaviour of a program. A collection of test cases is a 'test suite', representing that a certain section of the system has a specificied set of behavoiurs. A relevant example would be for a table in a database. The test suite would represent if the table has the desired validations in place and would consist of test cases that tested each specific behaviour in isolation. The tests would be run as a set to confirm the table has the desired behaviour. When undesired behaviour is detected the test result will highlight the exact error in the code. The design of these test cases is crucial. To design these cases information is required, this information is sourced in two main ways Functional and Structural.

Functional Testing also known Black-box testing, is the technique of creating test cases with information from a formal or informal functional specification. A functional specification is the description of intended program behaviour distinct from the program itself. The software requirements and or its design specification are most commonly used to derive the functional specification. The software entity under test is treated as a black box, the actual code implementation is not known, where proper inputs are fed in and the output is observed. If the output or behaviour is that specified in the specification the test has been passed, otherwise it has been failed. Example: When a user clicks the Home tab in the nav bar they are directed to the Home page, in this

case the input is the user clicking the home tab and the desired behaviour is being directed to the home page. Black box testing can detect some faults that white box cannot, such as absent behaviours that are in the functional specification of requirements but not coded into the software application in error. The systematic nature of Black-box testing can help avoid missed test cases and provide more consistent coverage.[5][6]

One approach to functional testing is a systematic approach. A systematic approach has four steps :[6]

1. Partition the functional specification into independently testable features using a divide and conquer approach. For example, for a database table, dividing a table in a databse into its fields, then dividing again into the properties of each field.
2. For each independently testable feature find a representative class of values or derive a model to test it. For a string field with property of length greater than five. We may derive a specail case of blank string, a string of length less than five and string with length greater than five.
3. Generate test case specifications. Finding concrete values for the reprensentive class of values or model above. Building on the previous example we may have { "", "less" ,"longer"}
4. Generate test cases and institute tests, turning specifications into tests and instituting them.

Structural testing, also known as White box testing. uses the physical implementation of the software itself such as source code as information to produce test cases. A common approach to white box testing is 'Control Flow' testing. Due to the nature of code varying greatly between projects and the complexity and detail that can be involved the following steps have been simplified whilst maintaining the core principle.

1. Identify a feature to be tested. This could be on a small or large scale, for our examples we will choose creating an entry on a database.
2. Create a flow graph plotting all steps and paths the feature can execute, this would include steps such as verification.
3. Identify paths through the flow graph, entry is created successfully and

displayed or unsuccessfuly and prompted with an error message.

4. For each path write a test case that is expected to execute this path, with valid variables an entry should be created and it should be displayed.

Software testing is necessary and very costly. "In a typical programming project approximately 50 percent of the elapsed time and more than 50 percent of the total cost were expended in testing the program or system being developed"[2]. Reducing the costs, both time and monetary, is the main motivation for Automated Testing. Another overlooked and unappreciated benefit of automating testing is that test case generation is one of the most intellectully demanding and critical challenges in software testing.[7] By automating this process it not only reduces costs but also allows developers to dedicate more time and effort to other areas, also in some cases it is harder to create a test case but easy to verify a generated test case is correct. A whole systems tests do not have to be automatically generated to reduce costs and benefit.

An Automated functional testing approach follows the same methodology as manual systematic generation of test cases. Each step is automated and follows the same principles. The information as before used to derive test cases is the functional specification. The functional specification is however only formal and has specified syntax so that it can be interpreted by the system that will generate the test cases. There is more creativitivy and design put into the functional specification as the test designer is usually limited to a choice of test selection criteria. For step 3 the system must also include value generation that can meet the representative class of values outlined in step 2.[6]

Automated functional testing tends to be more complex, as it has to understand and interpret human written code. One method that has recieved alot of attention from researchers and is Symbolic Execution. This approach is similar to that of Control Flow, where symbolic values, instead of concrete values, are used instead for program inputs. Programs variables are described by the symbolic expressions of those inputs. The state of the program includes the symbolic values of program variables, a program counter and the path contraint on symbolic values: a boolean formula over the symbolic values input. Using this method it can explore all possible path divergences through a system and identify stop points, where the path ends. The major

problem with automated White-Box testing is identifying if a behaviour, stop point or a specific divergence in Symbolic execution, is desired or not. This problem is known as the Oracle problem, as desired behavoiur of code is contained within its specfication and design, not its implementation. Therefore some level of user input is required.[7]

Another relevant challenge for Symbolic Execution is developing a system that can cover multiple languages at once. This is very complex and producing a system can produce feasible output can be impossible due to the path divergence problem, where, either a user has to specify so many models automation is not feasible, or it doesnt find a signifcant amount of feasible program paths. As Ruby on Rails design environment can vary drastically between projects due to the flexibility of its framwork and use of Gems I will only consider and that some level of User input is required I will only consider Black Box testing techniques when I come to designing the project. This will deliver a product that will be usable to a wider audience as it is dependent upon on specification for which I can define. Interpreting multiple languages and being flexible enough to be useful is out of the scope of this project.

PARAGRAPH SAYING BB CHOSEN SO VALUE GENERATION NEEDS TO BE CONSIDERED WHEN CHOOSING TOOLS

2.2 Testing in Rails

Ruby on Rails applications are primarily developed using the generation of skeletons with variables set by the developer. The skeletons save a vast amount of time by generating a default environment which can then be built upon according to the section under development. By default one of the files created is a test file and it uses the Ruby on Rails MiniTest class.[8]

TEST UNIT

RSPEC

CONCLUDE RSPEC

RSPEC IN GREATER DETAIL (may be few par)

TEMPLATES FOR RSPEC - NEED STRING MANIPULATION ETC

FACTORY GIRL

2.3 Tools to use

TOOL REQUIREMENTS FROM PREVISOUS SECTIONS

An MVC web application fits all these criteria. MVC, Model-View-Controller is an architectural pattern that seperates an application into three interconnected parts. This separation (of the three components) allows for responsibilities to be allocated independently to each component, seperating the logic from the user. The model is responsible for the data of the application and the rules and logic used to create and update the information. The view is responsible for displaying the data and possible interactions with the system to the user. The controller is responsible for controlling the flow of the system, accepting user input and converting it into commands for the model and view.

An example of the components interacting would be creating an entry to a database. The "view" would be responsible for displaying the form in which to fill in. On submission the "controller" will process the information, ensure only persmissable information is submitted and enter additional information, then send it to the "model". The model will verify the structure of the information, ensure the correct fields are present and accord with its rules. The model will then notify the controller if the submission was successful or not and the controller will update the view to reflect the status.

This seperation of components means that all user interaction with the database has to go through the controller and is therefore limited to what the developers want users to be able to do. This provides a high level of security as each action is controlled and the internal structure and representation of the information within the database is hidden. Simultaneous development is also possible due to the seperation of the components, work on the front and back end concurrently. Although I will not be able to get the full benefit of this aspect (as I am developing the project solo), it will allow me to shift focus as components do not need to be finished when switching from one to another, giving greater flexibility in development.

High levels of cohesion are inherited automatically from the architecture with the grouping of logically similar elements, this makes the code easier to read and creates a more natural flow within the source code. There are however some drawbacks to MVC architecture, they are inherently more complex due

to the seperation and the framework must be learned. Also there may be multiple programming languages involved. This steep learning curve could mean a large initial investment (time/cost) into a development team to learn a new framework and new languages.

MVC web application frameworks have become extremely popular and are behind some of the most used and powerful websites. Django an MTV, follows MVC architecture but its creators decided to rename the components [9] to better suit them, is behind the two most visted websites in the world Google and YouTube[10][11]. Ruby on Rails another MVC is behind Twitter, Airbnb and Soundcloud.[12] MVC frameworks are known for their scalablity, being suitable for the smallest to the largest projects. However FaceBook decided that its scalablity had reached its limit, that adding new features made the code exponentailly more complex.[13] My project will be no where near the scale of facebooks sourcecode so I do not need to worry about reaching the end of its scalibilty.

The chosen MVC to construct the project in is Ruby on Rails. I have done previous projects in both Ruby and the Rails framework, the rest of this section will show that Ruby on Rails is an adequate choice.

Ruby was selected as the primary programming language by default as it is the language that runs Ruby On Rails. Ruby is a dynamic, multi-paradigm programming language. The paradigms consist of Object-oriented, Imperative, Functional and Reflective making it a very powerful and versatile language. This combination is from its founder Yukihiro Matsumoto who was influenced by Perl, Smalltalk, Eiffel, Ada, and Lisp.

Rubys primary design goal was to make a language that he himself enjoyed using, by minimizing programmer work and the potential for confusion(Ruby Wiki). Achieved with a focus on human interaction, how programmers code and design applications as opposed to focusing on how the code will run on machines. And also following the principle of least astonishment, where the behaviour of the language minimizes confusion for experienced users.

The above two images show C++ 1 and Ruby 2 printing Hello world to the console. The comparison between the two languages highlights the efficiency and simplicity of the Ruby language.Ruby on Rails projects are commonly

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

Figure 1: C++ print Hello world to console

```ruby
puts "hello world"
```

Figure 2: Ruby print Hello world to console

worked on by a group of people and in multiple languages, therefore the simplistic syntax gives greater clarity and understandability to programmers who are lesser experienced in Ruby.[14]

Ruby is open source, free and redistributable with a vast range of existing code from both Ruby and its large community. Primarily consisting of Gems, code packages that can be installed and supported into a project easily and with minimal effort via RubyGems, and frameworks, such as Ruby on Rails. Making it very popular for education and business. Following the DRY Don't Repeat Yourself principle in a very effective and efficient manner.[14]

Ruby was ranked ninth on TIOBE index[**?**] and has become a very popular and respected language relative to its age among the other languages on the index. No alternatives could be considered due to the dependency of Ruby on Rails on Ruby, however Ruby is a very strong and durable language so it does not detract from the overall project.

FORMS AND DATABASE SUPPORT FOR INPUTTING INFORMATION THAT CAN BE PROCESSED

STRING MANIPULATION FOR TEMPLATES

NUMBER GENERATION

STRING GENERATION FROM REGEXP USING GEM REGEXP.RANDOM

RUBY HAS NECESSARY LOGIC TO CONTROL FLOW ETC

2.4 Evaluation

HOW TESTS ARE EVALUATED

HOW OTHER PEOPLE EVALUATED PROJECTS

INJECTION

DOG FEED

COMPARE OUTPUT WITH WHAT EXISTING HAD

COMPARE CODE COVERAGE

TIME SAVED

3 Chapter 3: Requirements and Analysis

This project started with the motivation of bringing automated testing to Ruby on Rails. From research carried out and disscussed in Chapter 2 and considering the scale of the project the way to achieve this was to focus on validation in the model component and produce RSpec test cases. A more precise description of the project is therefore **To create a system that automatically generates RSpec test cases for model validation in Ruby on Rails Applications**. To accomplish this aim there are three main aspects to the system, as follows:

1. Allow a user to create and maintain a formal specifcation

   A user should be able to describe all tables, fields and corresponding validation properties using the system. The user should also be able to edit and update the formal specification due to their changing demands.

2. Generate a valid RSpec test case

   The RSpec test case should be generated with information entered from defined formal specification. To be valid the RSpec test case must meet certain critera. The test case must run the same as a manually produced user test case. It must only test the specified behaviour, e.g Employee table, Age field, must be greater than 18, and nothing else. It must produce a human readable test case with a human readable behaviour descriptor that makes the user aware of exactly what behaviour has failed. It must produce a generated value that isolates the behaviour under test, it generates a value that fails the validation under test while passing the other validations for the field.

3. Consolidate all RSpec test cases for to make a valid test suite for a table

   All test cases for a table, for all its fields and assigned validations, must be consolidated into a test suite that runs as a manual written test suite.

To make the system viable and useful to developers there are some additional considerations

1. Ease of use

   The system built should be easy and intuitive to use.

2. Time saving

3. Creating the formal specification must take less time than it takes to write the tests manually

Ruby on Rails has fourteen data types supported natively be ActiveRecord[15]. Ruby on Rails also has many active record validations supported natively. To fit the scope of the project not all data types and validations will be supported. The selection of data types and validations supported are as follows.

Data Types Supported

1. Integer
2. Float
3. String

Integer and Float Validations

1. Greater than
2. Greater than or Equal to
3. Equal to
4. Less than or Equal to
5. Less than
6. Other than
7. Divisible by
8. Blank
9. Inclusion
10. Exclusion

String Validations

1. Maximum Length
2. Minimum Length
3. Exact Length
4. Format
5. Blank
6. Inclusion

Table 1: Formal Specification Requirements

ID	Requirement	Priority
1	A user can create a Project	**M**
2	A user can edit a Project	**M**
3	A user can delete a Project, associated Tables are also deleted	**M**
4	A user can create a Table, only for a given Project	**M**
5	A user can edit a Table	**M**
6	A user can delete a Table, associated Fields are also deleted	**M**
7	A user can create a Field, only for a given Table	**M**
8	A user can edit a Field	**M**
9	A user can delete a Field, associated validations are also deleted	**M**
10	A user can assign a validation, only for a given Field	**M**
11	A user can only create a value for a assigned validation	**M**
12	A user can edit a value	**M**
13	A user can view all Projects	**M**
14	A user can view all Tables associated to a given Project	**D**
15	A user can view all Fields associated to a given Table	**D**
16	A user can view all validations and values associated to a given field	**D**

7. Exclusion

Table 2: RSpec Test Case Requirements

ID	Requirement	Priority
1	RSpec test case should be runnable	**M**
2	RSpec test case should only test one behaviour	**M**
3	RSpec test case must should test behaviour intended	**M**
4	RSpec descriptor must be human readable	**D**
5	When test case fails, its output must specify exact behaviour at fault	**D**
6	RSpec test case should have human readable syntax	**D**
7	Be able to generate an Integer that satisfies all validations and their values assigned to a field	**M**
8	Be able to generate an Integer that does not satisfy a validation but satisfies all other validations and their values assigned to a field	**M**
9	Be able to generate a Float that satisfies all validations and their values assigned to a field	**M**
10	Be able to generate a Float that does not satisfy a validation but satisfies all other validations and their values assigned to a field	**M**
11	Be able to generate a String that satisfies all validations and their values assigned to a field	**M**
12	Be able to generate a String that does not satisfy a validation but satisfies all other validations and their values assigned to a field	**M**

Table 3: RSpec Test Suite Requirements

ID	Requirement	Priority
1	RSpec test suite should be runnable	**M**
2	RSpec test suite should be contain all test cases for a table	**M**
3	RSpec test cases should be grouped via field	**D**
4	RSpec test cases should be in a logical order	**D**
5	RSpec test suite must be available for download	**M**

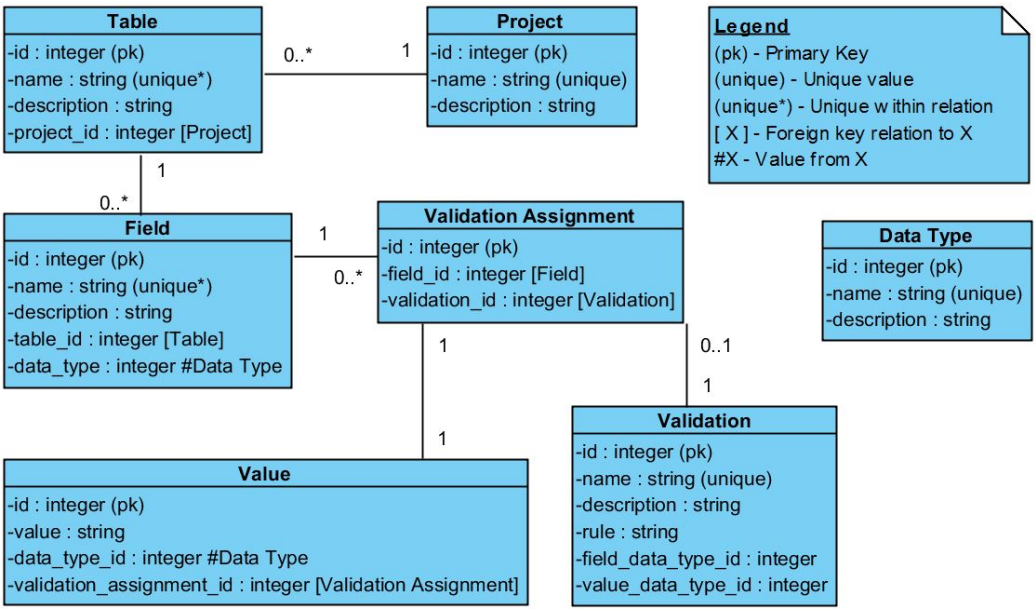

Figure 3: UML Database diagram

4 Chapter 4: Design

4.1 Database

The database was designed to accomodate the formal specification and extra information needed for RSpec test cases. Ruby on Rails has native support for relations via ActiveRecord, this provided flexibility in being able to have many relations. The design paradigm of sole responsiblity was carried forward and each table has one purpose.

Project

Project allows for a user to use the system for multiple projects, that is to organise groups of tables that are disconnected. The name field is unique to avoid confusion amongst projects. The description field is only for the user to add extra information if they desire and serves no functional purpose in the system. A project can have zero to many tables. When a project is deleted its associated tables should also be deleted.

Table

Table is the table in a user's database for which they are constructing a functional specification. A table can only be created with a valid project_id

foreign key. The name is unique within the scope of the project, multiple tables can exist with the same name within the database but they must belong to seperate projects. The description field is only for the user to add extra information if they desire and serves no functional purpose in the system. A table belongs to a project and has zero to many fields. When a table is deleted its associated fields should also be deleted.

Field

Field is the field in a user's database for which they are constructing a functional specification. A field can only be created with a valid table_id foreign key. The name is unique within the scope of the table, multiple fields can exist with the same name within the database but they must belong to seperate tables. The description field is only for the user to add extra information if they desire and serves no functional purpose in the system. Data_type_id is the fields data_type and is limited to values in the Data Type table, however no relation is forced. A field belongs to a table and has zero to many Validations through Validation Assignments. When a table is deleted its associated validation assignments should also be deleted.

Validation

Validation is a validation the user can associate to a field. The user can not create, edit or destroy these and they are seeded in the database. The name is unique within the scope of its field_data_type. The description field is to aid the user in understanding what the validation is. The rule field is used within the system to generate values. Field_data_type_id and value_data_type_id limited to values in the Data Type table, however no relation is forced, and are used so only appropriate validations can be assigned to fields. A validation has zero to many fields through validation assignments.

Value

Value is the value of a validation that a user associates with a field, E.g false for blank or ten for minimum length. A value can only be created with a valid validation_assignment_id foreign key. The value is stored as a string and the data type of which the system should treat the value as is stored in data_type_id, which refers to the data type table, but no relation is enforced. A value belongs to a validation assignment.

Validation Assignment

Validation assignment associates a field with a validation and also the value for that validation. The assignment is created then a value is created belonging to the assignment. A validation assignment can only be created with

both valid field_id and validation_id foreign keys. A validation assignment has one field, one validaiton and owns a value.

Data Type

Data type is used to provide consistency throughout the system by checking data types are equal between entities, checking values of are the correct type and also used in forms to reduce options available to the user. The user can not create, edit or destroy these and they are seeded in the database. The name field is unique to avoid confusion amongst data types and avoid possible duplication. The description field is to inform the user on the data type. Data type has no relations, but is used throughout the system in reference.

4.2 View and Flow

The main consideratons when designing the look and how the user navigates and uses the system were human-centered design principles with the goal of increasing effectiveness, efficency and satisfaction[16] .The principles taken into consideration and how they were applied are[17]:

1. **Simplifying the structure of tasks**

   An average user is able to remember five things at a time. Providing consistency within similar methods such as creation of entitys and also clearly displaying where in the system the user is reduces the strain on both long and short term memory. The creation, deletion and editing of entitys follows the familiar and consistent method of filling in a form and submitting it. At the top of each page it clearly displays how deep in the system and also the names of each level. This means the user will have to remember less when using the system.

2. **Exploiting the power of constraints**

   Reducing and not exposing a user to redundant or irrelevant information allows a user to use a system more efficently and with less effort, by not having to process such information. This is used by displaying only the related entities when viewing an entity, only the fields for a table are shown when viewing that table. When a user is creating an entity with a form options are reduced to those that are valid, only choose from Integer validations for a Integer field. Entitys that are dependent on a another, field is dependent on table, are only available to create when viewing the dependent upon entity.

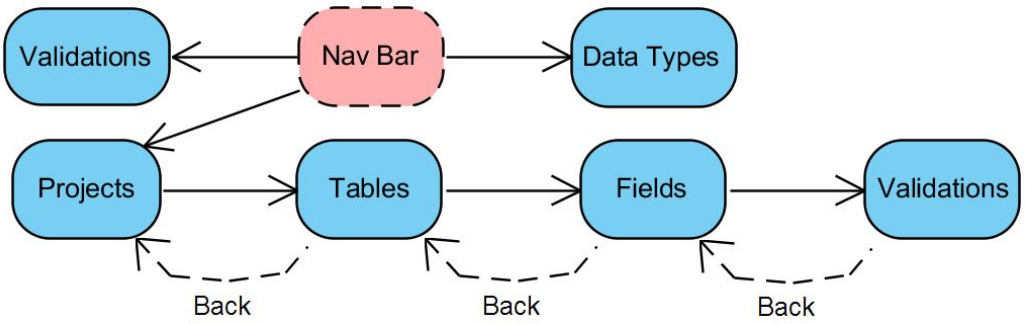

Figure 4: The navigational flow of the system

3. **Make things visible**

   Bridging the gulf of execution, buttons and links do what a user expects them to do. This principle followed by using standard naming conventions on standard objects such as buttons and links that a user is familiar with.

General

Consistency was at the the forefront of the visual and flow design. Each page that has a dependent entity can only be created via a button on the page of that entity6. When viewing an entity it will display all and only its dependent entities in a table.6 The dependent entites can be viewed, edited and deleted from this table via links in the row of that entity on the table7. Each page that is dependent upon on entity can access the dependent enity via the back button7, also applies naviagtionally to forms8. The header at the top of each page displays the depth the user is at by underlying the current level while also indicating the levels with the name of the entity at that level7.

Nav bar

The nav bar is not a page but the navigational bar displayed at the top of each page5. It links to the projects page, validations page and the data types page via the appropiatley name button.

FUNCTIONALITY OF PAGES ???

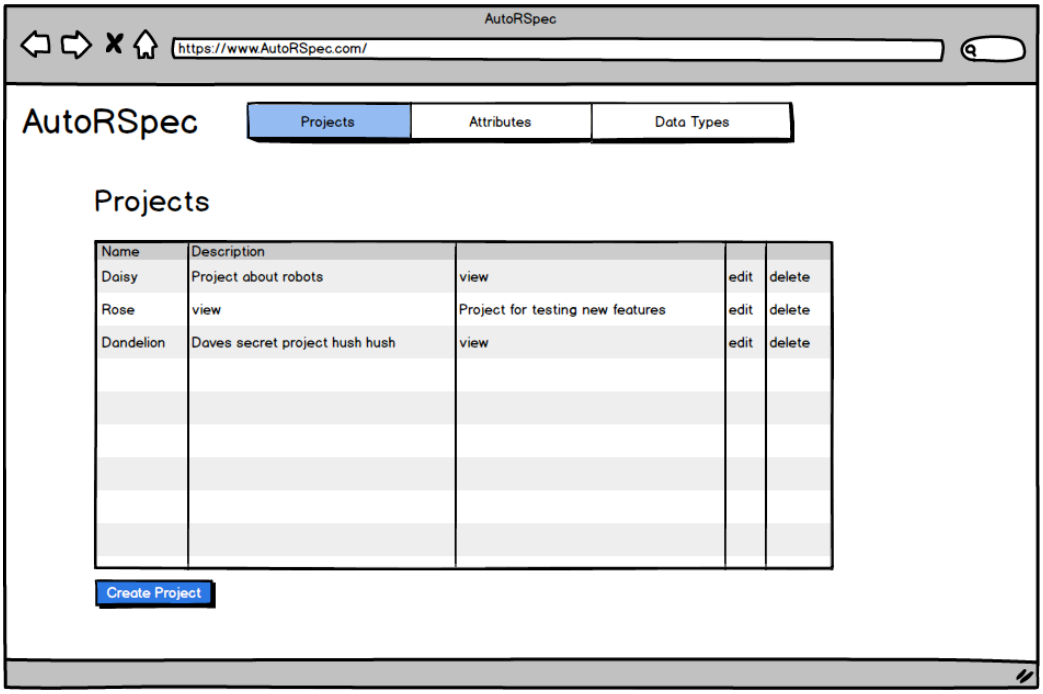

Figure 5: Mock up of projects page for the system, also home page.

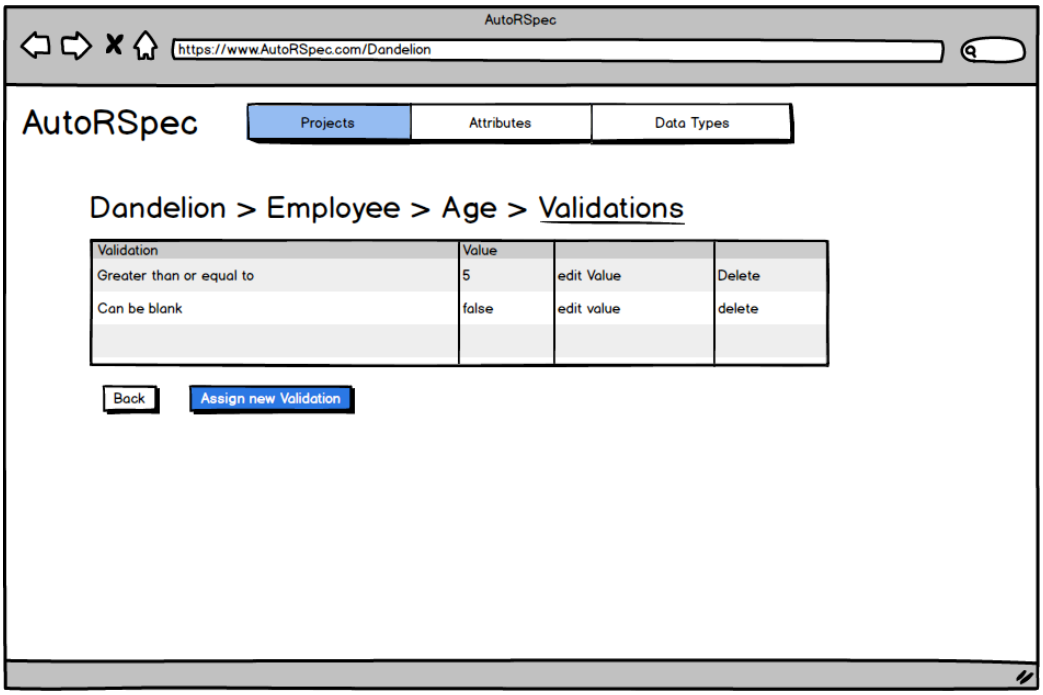

Figure 6: Mock up for a table in the system displaying its fields

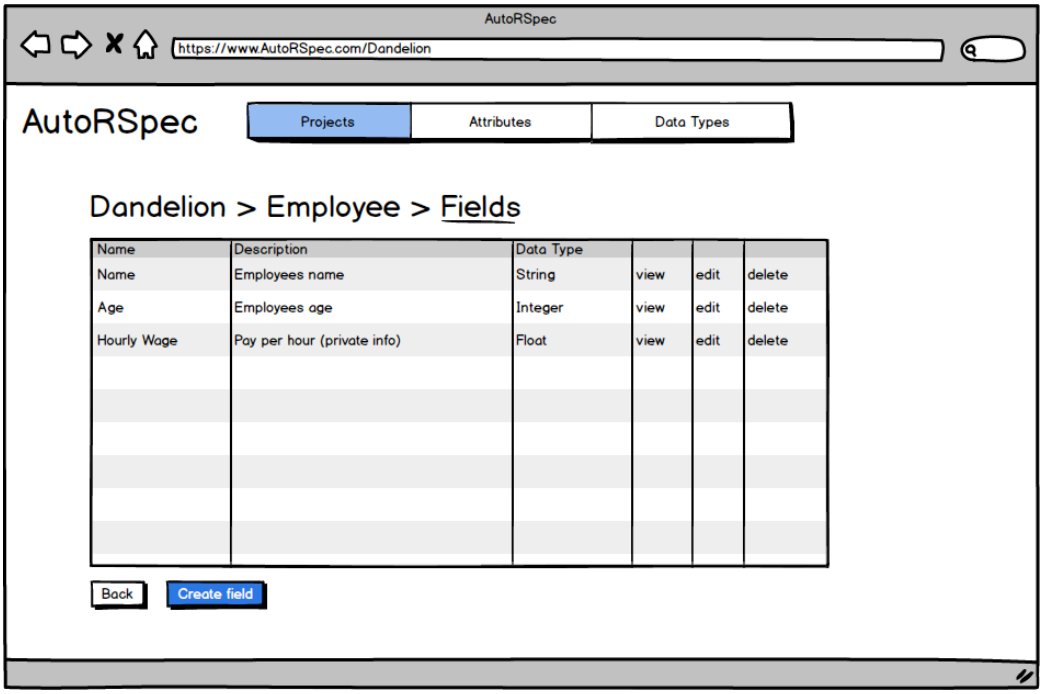

Figure 7: Mock up for a field in the system displaying its validation

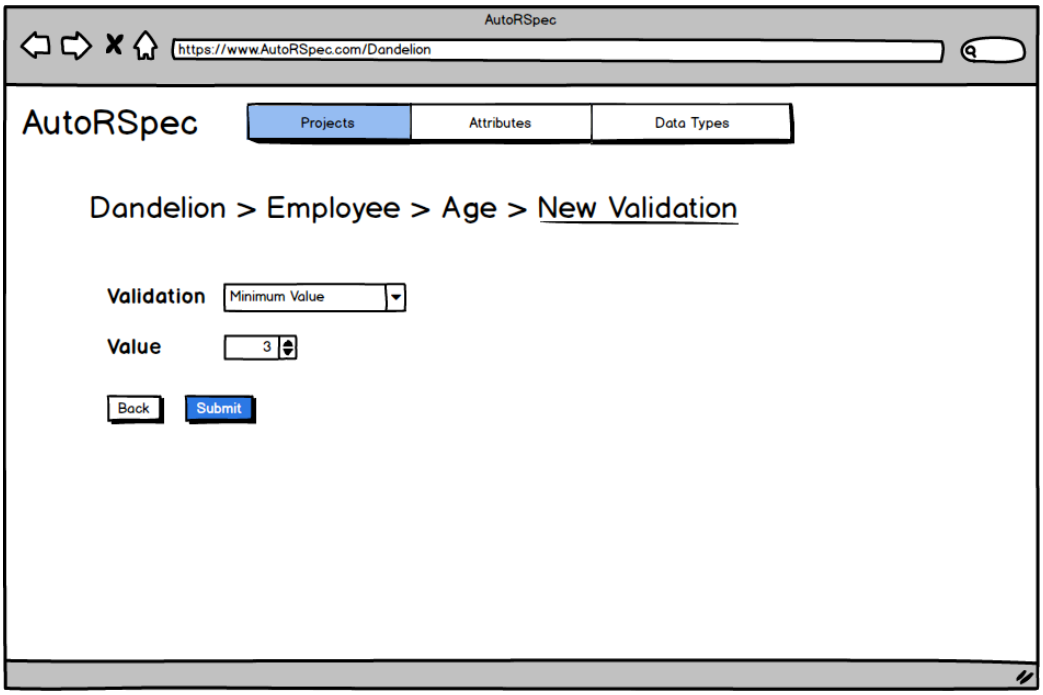

Figure 8: Mock up for a form in the system

4.3 Test Case Generation

4.3.1 Value Generation

Listing 1: Pseudo code for value generation label

```
1  generate_value( isolatedValidation, validations)
2   for iterationCount in 1 to X
3    randomValues = generate_random_values Y
4    if isolatedValidation not nil
5     randomValues = fail_validation(isolatedValidation,
          randomValues)
6     if randomValues is empty
7      next
8    for validation in validations
9     randomValues = pass_validation(validation,
          randomValues)
10    if randomValues is empty
11     next
12   if randomValues is empty
13    next
14   else
15    return random value from randomValues
16  return ERROR NO VALUE GENERATED
```

The psuedo code in Listing 1 shows then general approach to generating a value for a test case. To increase the efficency and speed of potentailly returning a valid value by generating **X** values at a time for a potentail of **Y** times. Generating **X*****Y** values initailly will take a greater deal of time and may be overkill. The correct balance will need to be found in implementation of the initail amount of values for each iteration and how many iterations to carry out.

The function will take two parameters, a validation to fail and a list of validations to pass. The validation to fail can be nil, in which case it will produce a value that passes all validations in validations parameter. The validations parameter can also be an empty list in which case it will return a value that fails the isolated validation. In the case of nil validation and and empty list of validations or no value can be generated it will return an error.

The flow of the function is to generate a random amount of values, then re-

move those values that pass the isolated validation, afterwards it will remove those that fail each validation in validations. After each time this array of random values is manipulated it will check if its empty, if so it will skip to the next iteration, this improves speed and efficency by eliminating unnesscary operations. When all validations have been processed it will return a random value from random values. The general flow and principles is the same for each data type that values will be generated.

4.3.2 RSpec Test Case

Listing 2: Pseudo code for value generation label

```
1 it "is (invalid with a value that is not VALIDATION
    VALVALUE" do
2   TABLENAME = build(:TABLENAME, FIELDNAME:
      GENVALUE
3   if TABLENAME.respond_to?(:valid?)
4    expect(TABLENAME.not_to be_valid, lambda
       TABLENAME.errors.full_messages.join("\\n")
5  end
6 end
```

VALIDATION is the validation that is being tested on the field, and is described as its polar opposite

VALVALUE is the value of the validation that is being tested on the field.

TABLENAME is the name of the table the field belongs to

FIELDNAME is the name of the field under test

GENVALUE is the value that will test the validation

The RSpec test case template that will be populated with information from the functional specification and value generated. An example of a employees age field with validation of greater than value eight with a generated value of thirteen.

Listing 3: Pseudo code for value generation output label

```
1 it "is invalid with a value that is not greater than 8"
    do
2   employee = build(:employee, age: 13)
```

```
3    if employee.respond_to?(:valid?)
4     expect(employee.to_not be_valid, lambda employee.
          errors.full_messages.join("\\n")
5   end
6  end
```

4.3.3 RSpec Test Suite

The test suite is responsible for the overall document structure that the user downloads. It will provide the preamble, then iterate through each field and finally end the document. It will provide all necessary logic to navigate test cases correctly and add the correct syntaxical links to the file.

Listing 4: Pseudo code for value generation label

```
1  require "spec_helper"
2
3  desrcibe "TABLENAMECAP" do
4   it "has a valid factory" do
5    expect(build(:TABLENAME)).to be_valid
6   end
7   #for each field belonging to table |f|
8    describe "f.NAME has property" do
9    # genval = generate_value(nil, f.validations)
10   # add to file test_case_pass_all(f,genval)
11   #for each validation belonging to f |v|
12    # genval = generate_value(v, rest of validations)
13    # add to file test_case(v,f,genval)
14   #end
15   end
16  #end
17 end
```

EXAMPLE ??

To make the pseudo code more clear lines starting with # are not printed to the document but code that is ran. TABLENAMECAP is the table name capitalised. Lines 1-4 are the preamble and set up the RSpec document and write a test to check the tables factory is valid. Then for each field it will add the correct syntax to the file for that field and follow it by adding a test case to the file that passes all validations. For each validation that field has been

assigned it will iterate through isloting each field and adding the relevant test case to the file. Test suite is the top level function therefor before it calls to generate a test case, lines 8,11, it will call the function to generate a valid value for that test case, lines 7, 10. Test suite will finally end the document with the appropiate "end" syntax that it opened in the preamble.

5 Chapter 5: Implemention and Testing

5.1 Setup

Ruby version 2.1.7 and rails 4.2.5.2 were used to construct this project.

Github[18] was used throughout the project to provide version control and to prevent any catastrophes. The airbnb ruby style convention was used to provide clean looking and consistent code[19]. Rails new was used to instatiate the system. Then the generate scaffolfd command was used to construct the base for each table in the database schema outlined in chapter 4.1 9. This sets up the default mvc components for each, the exact files created can be seen in Figure 9.

Regular expressions are used frequently through the implementation process, to quickly check a regular expression was serving the function expected the Rubular[20] website was used to quickly verify them, this was not for testing but to provide further clarity on complex expressions. The gem Annotate[21] was used to annotate the models with database information such as field names to help efficency when implementing as the information is all in the same place. The test suites that the system produces are dependent upon the user having RSpec and factoryGirl installed, in addition the factories also have to be set up.

5.2 Database

Rails natively supported active record assocations were used to establish the relations between the tables in the database[22]10. The dependent destroy option was used on relations where the deletion of the entity desired the deletion of entites that depended upon it outlined in the requirements, this will cause a cascade effect downwards through relations10. Validations were applied intially according to the requirements then also dynamically throughout the process as they came relevant through implementation, e.g a fields name

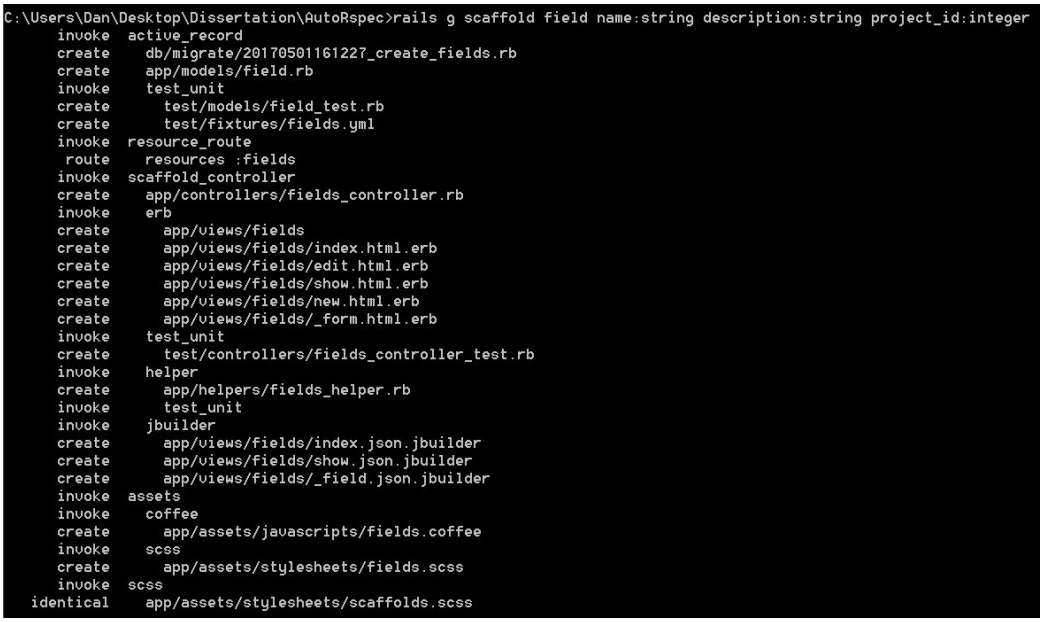

Figure 9: Rails g scaffold command for field

```
belongs_to :table
has_many :validation_assignments, dependent: :destroy, inverse_of: :field
has_many :validations, through: :validation_assignments
```

Figure 10: Rails native relation methods used in field.rb model

should follow the lower case underscore format that is convention in naming table fields in Ruby on Rails applications**??**. These validations were for the functionallity of the test suite generation and related functions.

```
validates :name,
  format: {with: /\A[a-z][a-z_]*[a-z]\Z/, message: "Must follow lower case underscore naming convention"},
  presence: true,
  uniqueness: {scope: :table_id, message:"Field already present in table"}

validates :table, presence: :true

validates :data_type_id, presence: :true
```

Figure 11: Rails native validation method used in field.rb models

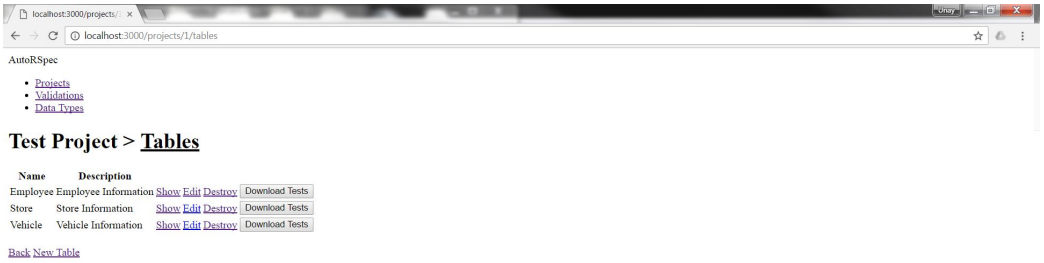

Figure 12: Tables page without bootstrap

5.3 View and Flow

Bootstrap[23] was used for the css of the web pages. Bootstrap gives a very clean and presentable wesbite by adding default css to HTML elements. Further classes that were used were table, table-hover, button classes and navbar classes. Bootstrap is extremely efficent for the view aspect of implementation.

To get the flow outline in design the routes file had to be heavily configured. Rails reads the routes file from top to bottom, on the first path it hits that a match it will route to. Placing the custom routes above the nesting resources effectively overrides the defaults that rails will produce. The defaults needed to be overrode due to the nature of how when viewing an enity it displayed all of its dependent entities, aswell as being able to create a dependent entity. When the routes file had been configured rails it created the paths that could be used throughout the project15. The path helpers are desired as they take variables and produce a link thefore aid in the scaling and readablity of the project whilst also providing a very clear and methodical link.[24]

5.4 Test Case Generation

5.4.1 Value Generation

The design for generating a value was modified slightly due to finer details but its overall flow and princple was kept. One oversight from the design was the exclusion validation, when an exculsion validation is to be isolated

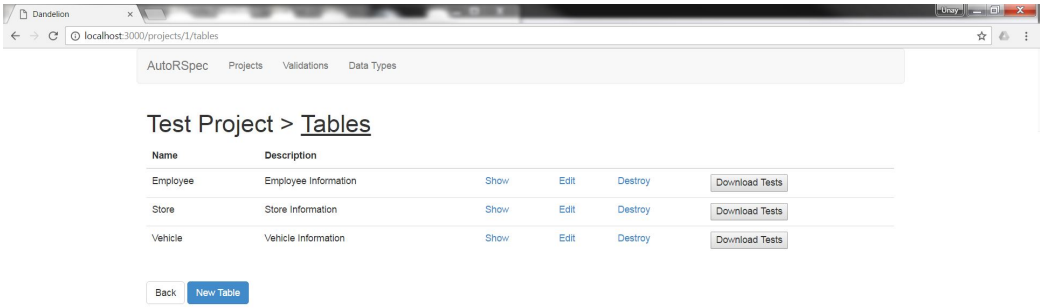

Figure 13: Tables page with bootstrap

```
resources :validations
resources :data_types

#Button for generating tests
post "tables/download/:project_id/:id" => "tables#download"
#new
get 'projects/new' => 'projects#new'
get 'projects/:project_id/tables/new' => 'tables#new'
get 'projects/:project_id/tables/:table_id/fields/new' => 'fields#new'
get 'projects/:project_id/tables/:table_id/fields/:field_id/validation_assignments/new' => 'validation_assignments#new'
get 'projects/:project_id/tables/:table_id/fields/:field_id/validation_assignments/:validation_assignment_id/values/new' => 'values#

#show => index
get 'projects/:project_id' => 'tables#index'
get 'projects/:project_id/tables/:table_id' => 'fields#index'
get 'projects/:project_id/tables/:table_id/fields/:field_id' => 'validation_assignments#index'


resources :projects do
  resources :tables do
    resources :fields do
      resources :validation_assignments do
        resources :values
      end
    end
  end
end
```

Figure 14: Routes file for project

```
<%= link_to 'Back', project_table_fields_path(@project, @table, @field), class:"btn btn-default" %>
<%= link_to 'New Validation assignment', new_project_table_field_validation_assignment_path(
  @project, @table, @field), class: "btn btn-primary"  %>
```

Figure 15: Example of path helper

it is logical to just return a random value from the list of values excluded as they will all fail the validation rather than try to randomly generate a value in the list.

String Generation

The generate_string_value function9 is the top level function responsible for generating a string value for a test caseTo explain the function in depth the functions it uses will be explained first.

Listing 5: gen_rand_string_array

```
1  def gen_rand_string_array(regex, amount)
2    array = []
3    # Generate amount specified as generated empty
         strings are deleted
4    while array.length < amount
5      string = regex.random_example(max_repeater_variance
           : 50)
6      if string != ""
7        array << string
8      end
9    end
10   return array
11 end
```

gen_rand_string_array5

gen_rand_string_array takes two parameters regex and amount and returns an array of strings. regex is the regex from which the strings will be generated from and amount is the amount of strings to return in the array. The actual string value is generated using the 'regexp-examples'[25] gem. The gem extends the regexp functionality by adding the function of generating a random string that is valid for that regular expression. The max_repeater_variable is set to 50, setting the amount of characters that can be included in the generated string output from each subsection of the regular expression pattern to be in the range of zero to fifty. The generated string can be an empty string so the generation is wrapped in a while loop to ensure the amount of strings returned is that of those requested in the parameters.

Listing 6: pass_string_rule

```
1  def pass_string_rule( strings, rule, value )
2    if rule == "regex"
```

```
3       regex = value.to_regexp
4       return strings.keep_if{ |n| n =~ regex } # =~ , ""[
            regex] , "".match(regex) #ATTN TIMINGS
5     elsif rule == "exclusion"
6       excluded = value.split(",").map(&:strip)
7       return strings.delete_if{ |n| excluded.include?(n)}
8     else # ASSUMED: Only length validations can progress
          to here
9       # Comparorator stored as string in database e.g ">"
10      return strings.keep_if{ |n| n.length.method(rule).(
            value.to_i)}
11    end
12  end
```

Listing 7: fail_string_rule

```
1   def fail_string_rule(strings, rule, value )
2     if rule == "regex"
3       regex = value.to_regexp
4       return strings.delete_if{ |n| n =~ regex }
5     else # ASSUMED: Only length validations can progress
          to here
6       # Comparorator stored as string in database e.g ">"
7       return strings.delete_if{ |n| n.length.method(rule)
            .(value.to_i)}
8     end
9   end
```

pass_string_rule and fail_string_rule

pass_string_rule takes three paramaters strings, rule and value and returns an array of strings. rule is the validation with value being the value of the validation, E.g rule = minimum length and value = 4, strings is the array of strings that have to pass the validation to be returned. The function starts by chechking if the rule is format validation, known internally as regex, if so it will remove any string form strings array parameter that does not meet the regular expression specified in the value parameter and return the array. Values are stored on the system database as strings so the 'to_regexp' gem[26] is used to convert the string into a regexp object.

Next it checks if the rule is exclusion, if so it will process the value parameter as seen on line 6 into an array of strings due to how its stored one string

on the database. Then it will return the array of strings passed as the parameter with any string that is in the exclusion array removed. After these two checks the only validations that can remain for strings are length validations. Inclusion and blank are handled in another function and rule cannot be the exclusion or format validation. rule for length validations is a comparorator (>=,>,==,<,<=) stored as a string in the database. This allows the functionallity of, return strings array parameter with strings kept if the the length of the string is 'rule' value. E.g keep if >3.

fail_string_rule follows the same format as pass_string_rule with keep and delete string from string parameter array switched. fail_rule_string however does not have to check for exclusion as this is handled in generate_string_value.

Listing 8: prep_string

```
1  def prep_string(string)
2    # Add speech marks and escape \ " within to be
         written safely to test file
3    return "\"" + string.gsub(/[\"\\]/){ |c| "\\"+c }+"
         \""
4  end
```

prep_string

prep_string takes a string and returns in with speech marks concated on each side and the '\' and ' " ' characters escaped. It is used so when the string value is printed to file it is in the correct format and doesn't have any undesired effects that would occur if the characters were not escpaed.

Listing 9: generate_string_value

```
1  def generate_string_value(isolated, rest)
2    # isolated can be passed as nil to generate a pass
         all
3    # default regexp to generate strings
4    regexp = [/\A[a-zA-Z]*[a-zA-Z]*\z/, /\A[\w]*[\w]*\z/,
         /\A[\w]*[[:punct:]]*[\w]*\z/,
5      /\A[\w]*[[:print:]]*[a-zA-Z]*\z/]
6    # Add users regexp to the defaults if in rest
7    if !rest.empty?
8      if validations_contain?("format", rest)
9        index = validations_index_of("format", rest)
10       regexp << rest[index][1].to_regexp
11     end
```

```
12    end
13    # Begin generation loop
14    for full_iteration in 1..50
15      strings = []
16      # If testing exclusion then return one of its
            values
17      if (isolated != nil) && (isolated[0] == "exclusion"
            )
18        return prep_string(isolated[1].split(",").map(&:
              strip).sample)
19      else
20        regexp.each do |r|
21          strings += gen_rand_string_array(r, (1000/
                regexp.length))
22        end
23      end
24      if isolated != nil
25        strings = fail_string_rule(strings, isolated[0],
              isolated[1])
26        if strings.empty?
27          next
28        end
29      end
30      rest.each do |rule|
31        strings = pass_string_rule(strings, rule[0], rule
              [1])
32        if strings.empty?
33          break
34        end
35      end
36      if strings.empty?
37        next
38      else
39        return prep_string(strings.sample)
40      end
41    end
42    # No valid value generated
43    return "\"!!!ERROR_GENERATING_VALUE\""
44  end
```

generate_string_value

The generate_string_value function9 is the top level function responsible for generating a string value for a test case. It takes two parameters isolated and rest. isolated is the validation that has been isolated and is under test in the test case. rest are the other validations that are associated to the field under test. When both parameters are present it will return a string value that violates the isolated validation and obeys those in rest, rest can be an empty list. If isolated is nil it will return a string value that obeys all validation in rest. If a string value isnt generated it will return the string on line 43.

At the begining of the function some regular expressions are defined in array, the regular expression decribe a diverse and varied range of patterns for strings. The function then checks to see if the validations to pass contain the format validation, if so it will concatenate the regular expression to those array defined previously. This is to include strings from the users format in the generation pool. After this check the iteration begins as described in the design chapter, the maximum iterations is fifty.

At the begining of the iteration an empty array is definfed call strings, this variable will contain the randomly generated strings to be processed. The first step of the iteration is to check if the isolated validation is 'exclusion', if so it will return a random tring from the array of excluded values stored in the database from the formal specifcation as they are the only strings that will fail the exclusion validation. This step could preceed the for loop but was included as it provides clarity having all validation handling within the iteraions, also it will only ever be ran once so no efficeny is lost. If exclusion is not the isolated validation then it will populate the strings array with randomly generated strings from the regular expressions 'regexp' defined above, generating 1000 strings in total with the same amount from each regular expression, giving wide range of diverse strings.

Once the strings have been generated the function checks if isolated is not nil, if so it will remove strings from the strings array that pass the isolated validation via fail_string_rule. Next it will iterate through each validation in rest and remove each string from strings array that fail the validation via pass_string_rule. Once all validations have been iterated through it will return a random value from strings array after preparing via prep_string. Throughout the function after the strings array has been modified, strings potentailly removed, it will check if strings is empty and if so will 'next' to the next iteration. This is because strings is empty so no valid string remains so to increase efficency and save time it is logical to go to the next iteration.

Number Generation

Generating values for the float and integer datatypes follows the exact same methodolgy and structure as string. Therefore the slight differnce will be highlighted and infered as changes to generate_string_value. Numbers are randomly generated using rubys Random.rand with a range of negative one million to positive one million. The pass and fail rule functions dont have to account for the format validation but do have to account for divisible, which is accomplished by a map across the array of numbers. Also instead of comparing the length of a string with the comparorator the actual number is compared10. The last differnce is that the number returned does not need to be prepared like the string values.

The validators equal to and other than were removed as they are supported by creating a instance of inclusion or exclusion with one entry respectively. They were removed to reduce information overload for the user and reduce conflicting options

Listing 10: Differnce in pass/fail num rule to string pass/fail

```
1 if rule == "divisible"
2   return numbers.keep_if{ |n| n % value == 0}
3 else # ASSUMED: Only >,>=,<,<= validations can progress
      to here
4   # Comparorator stored as string in database e.g ">"
5   return numbers.keep_if{ |n| n.method(rule).(value) }
6 end
```

5.4.2 RSpec Test Case

In the design the behavioural description line of the RSpec test case, it "has this behaviour" do, was orginally in the test case template. However this was removed and is now handled by the test suite section, the change was to provide a more consistent test cases overall and make give the code greater clarity. Origanlly alot of variables were passed in through the hash and it made the template more complex and reduced its clarity, where as handling in test suite this problem was averted.

test_text11

test_text is the template for all test cases apart from the specail case of blank test which is explained after this section. test_text takes three parameters

valid, hash and indent. valid is a boolean and is used to change the value of valid in the hash, this variable is a string that is repsonsible for the matcher "to" ot "to_not" and is set according to the boolean, lines 3-7. hash is a hash that contains the necassary information to populate the template. The last parameter indent is the indentation the test case should start at. The function returns the string with the parts wrapped by %{ and } replaced by the string value of the variable name in the hash. The hash is contructed in test suite and passed into test_text as a parameter.

Listing 11: test_text

```
1  def test_text(beValid,hash, indent)
2    tab = "\t"
3    if beValid
4      hash[:valid] = "to"
5    else
6      hash[:valid] = "to_not"
7    end
8    template = tab*indent + "%{tableName} = build(:%{tableName}, %{fieldName}: %{
       generatedValue})\n" +
9                    tab*indent + "if %{tableName}.respond_to?(:valid?)\n" +
10                   tab*(indent+1) + "expect(%{tableName}).%{valid} be_valid, lambda
                       {%{tableName}.errors.full_messages.join(\"\\n\")}\n" +
11                   tab*indent + "end\n"
12   return template % hash
13 end
```

blank_test12

The can be blank vaildator has its own template because the variations between the desired output and test_text template were to great to make it clear and functional. The function follows the exact same methods, except it has a differn't template and it also contains the behaviour.

Listing 12: blank_test

```
def blank_test(hash, bool, indent)
  tab = "\t"
  template = tab*indent + "it \"%{decision} be blank\" do \n" +
             tab*(indent+1) + "%{tableName} = build(:%{tableName}, %{fieldName}: nil)\n" +
             tab*(indent+1) + "if %{tableName}.respond_to?(:valid?)\n" +
             tab*(indent+2) + "expect(%{tableName}).%{equator} be_valid , lambda {%{tableName}.errors.full_messages.join(\"\\n\")}\n" +
             tab*(indent+1) + "end\n" +
             tab*indent + "end\n\n"
  if bool
    hash[:decision] = "can"
    hash[:equator] = "to"
  else
    hash[:decision] = "can not"
    hash[:equator] = "to_not"
  end
  return template % hash
end
```

5.4.3 RSpec Test Suite

Listing 13: generate_tests_for_table

```
1  def generate_tests_for_table(id)
2
3      table = Table.find(id)
4      fileName = table.name.downcase + "_spec.rb"
5      #file = File.open(fileName, "w")
6      file = Tempfile.new(fileName,'tmp')
7      # Introduction/Set-up file
8      file.puts "require_\"spec_helper\""
9      file.puts ""
10     file.puts "desrcibe_\"" + table.name.titleize + "\"
           _do"
11     file.puts ""
12     file << write_factory_test(table.name, 1)
13
14     #Context loop for each field
15     # Hash stores variables to populate test templates
16     hash = { tableName: table.name.downcase}
17     table.fields.each do |f|
18       hash[:fieldName] = f.name
19       if f.validation_assignments.empty?
20         next
21       else
22         file.puts "\tdescribe_\"" + f.name + "_field_
               has_validation\"_do"
23         file.puts ""
24         validations = package_validations(f.
               validation_assignments)
25         if validations_contain?("blank", validations)
26           index = validations_index_of("blank",
                 validations)
27           if validations[index][1] == "true"
28             file << blank_test(hash, true, 2)
29           else
30             file << blank_test(hash, false, 2)
31           end
32           # Remove blank from validations, fully
                 handled
```

```
33              validations.delete_at(index)
34              hash.delete(:decision)
35              hash.delete(:equator)
36            end
37            if validations_contain?("inclusion",
                validations)
38              index = validations_index_of("inclusion",
                  validations)
39              if DataType.find(f.data_type_id).name == "
                  String"
40                stringArray = validations[index][1].split("
                    ,").map(&:strip)
41                file << string_inclusion(hash, stringArray)
42              else
43                numberArray =
                    database_string_to_number_array(
                    validations[index][1])
44                file << number_inclusion(hash, numberArray)
45              end
46              next # Inclusion supercedes remaining
                    validations, skip to next
47            end
48            # test with all validations valid
49            file.puts "\t\tit \"is valid with a generated
                value\" do\n"
50            if DataType.find(f.data_type_id).name == "
                String"
51              value = generate_string_value(nil,
                    validations)
52              hash[:generatedValue] = value
53              file << test_text(true, hash, 3)
54            else
55              value = 0
56              if DataType.find(f.data_type_id).name == "
                    Float"
57                value = generate_value_integer(nil,
                      validations, "Float")
58              elsif DataType.find(f.data_type_id).name == "
                    Integer"
59                value = generate_value_integer(nil,
                      validations, "Integer")
```

```
60            end
61            hash[:generatedValue] = value
62            file << test_text(true, hash, 3)
63          end
64          # Write test isolating each validation to be
                invalid
65          validations.each do |p|
66            # Array we can manipulate
67            rest = [].replace(validations)
68            rest.delete_if{ |e| e == p} # remove isolated
                  rule
69            file.puts "\t\tit \"is invalid with value
                 that is not" + p[0].to_s + " " + p[1].
                 to_s + "\" do"
70            if DataType.find(f.data_type_id).name == "
                 String"
71              value = generate_string_value(p, rest)
72              hash[:generatedValue] = value
73              file << test_text(false, hash, 3)
74            else
75              value = 0
76              if DataType.find(f.data_type_id).name == "
                   Float"
77                value = generate_value_integer(nil,
                     validations, "Float")
78              elsif DataType.find(f.data_type_id).name ==
                     "Integer"
79                value = generate_value_integer(nil,
                     validations, "Integer")
80              end
81              hash[:generatedValue] = value
82              file << test_text(false, hash, 3)
83            end
84            file.puts "\t\tend"
85            file.puts ""
86          end
87        end
88        file.puts "\tend"
89        file.puts ""
90      end
91      file.puts "end"
```

```
92      file.close
93      return file.path, fileName
94    end
```

generate_tests_for_table13

generate_tests_for_table is the function that takes on the responsiblities of test suite outlined in the design section. It takes the id of the table for which to generate the test suite for as a parameter an returns the file path and file name. When a user clicks the download link for a table is will invoke the download function, this invokes generate_tests_for_table which using its return values the download function downloads the file to the user via the file path with the file name as file name.

The function begins by fetching the table from the database then creating the filename by taking the name of the table, downcasing it and concetaning '_spec.rb' to it. Then a temporary file is created to write all of the information to, it creates a temporary file as it not needed locally for long as the user will download it once it has been generated. It then writes the necessary require and RSpec syntax to introduce the tables tests , lines 7-11. After this the factory test is wrote to the file, this is a simple test that makes sure the factory girl factory for the table the user has created is functioning correctly. The table name is then saved to a hash that will be used to store information for the test case tempaltes, this completes the preamble.

After the preammle the tables fields are iterated through. At the begining of each iteration the fields name is saved to the hash, then if the field has no validations it will 'next' to the next iteration. If there are validations it will write the RSpec syntax to introduce the field, lines 22-23. Next it will package the validations into a more manageable format as oppose to that stored on the database. Once packaged it will check if the validations contain the blank validation, if so it will write the blank test to the file using blank_test and remove the validation from the packaged validations.

The next check is for the inclusion validation and if it is present the data type appropiate function is called,explained after this section, to produce test cases which are added to the file then it 'next' to the next iteration. It skips the rest of the validations as inclusion supercedes all validations except blank, which is why blank precedes inlusion in order of checks in function. The next part in the function is to write a test case for where all validations are considered, this is to provide the information that the generation can create a valid value for all validations therfore making the isolated validations valid

test cases, line 48-63. After it has been written to the file the next part is to iterate through each of the validaions.

At the start of each iteration rest, an array that contains all other validations is created. Then the behavioural descriptor is wrote to the fiel, E.g it "is invalid with a value not less than 5" do. Depending on the data type of the field the appropiate value generation is called and saved to the has before calling test_text which is written to the file. After each validation has been iterated through the function writes the correct syntax to the file to finish it then closes the file, then returns the path and filename.

Inclusion

Inclusion is handled seperatley as it supercedes other validations and has to be treated in a slightly differn't way due to the only values being accepted are those in the inclusion list. The possibilty of randomly generating a value that is in this list is too small to compute effeicently. Before calling the function generate_test_for_table will convert the value string in the database to an array of the relevant data type. Number and string inclusion are dealt with in the same manner so only string inclusion will be explained in this document.

string_inclusion 14 starts by storing in a variable a random value from the array of included values, this value will pass the validation. Then it will generate an array of random strings the same was as generate_string_value and remove any string from the array that is in inclusion, when a string is generated that is not included it is stored in a variable. Then it calls test_text with the pssing variable and the valid boolean set to true to get a passing test case, after it calls it again the failing value and false, lines 18-21. After this it will concenate the test cases with there behavioural descriptors, lines 22-24, and return the string to generate_test_for_table to be wrote to file.

Listing 14: generate_tests_for_table

```
1  def string_inclusion(hash, includedValues)
2    correctValue = prep_string(includedValues.sample)
3    incorrectValue = ""
4    regexp = [/\A[a-zA-Z]*[a-zA-Z]*\z/, /\A[\w]*[\w]*\z/,
         /\A[\w]*[[:punct:]]*[\w]*\z/, /\A[\w]*[[:print
       :]]*[a-zA-Z]*\z/]
5    strings = []
6    for i in 1..50
7      regexp.each do |r|
```

```
8         strings += gen_rand_string_array(r, (1000/regexp.
              length))
9       end
10      strings.delete_if{ |n| includedValues.include?(n) }
11      if strings.empty?
12        next
13      else
14        incorrectValue = prep_string(strings.sample)
15        break
16      end
17    end
18    hash[:generatedValue] = correctValue
19    correctTestText = test_text(true, hash, 3)
20    hash[:generatedValue] = incorrectValue
21    inCorrectTestText = test_text(false, hash, 3)
22    correct_intro = "\t\tit \"is valid with a value from
          inclusion\" do\n"
23    incorrect_intro = "\t\tit \"is invalid with a value
          not in inclusion\" do\n"
24    return correct_intro + correctTestText +
          incorrect_intro + inCorrectTestText
25  end
```

5.5 Testing

The first stage of testing was to set up the factories for the models using factory girl[27]. Factories allow for rapid and reliable testing using the once established factory to build a valid entity in test cases, seen on line 6 in Listing 16 . In fig 15 the factory for field is shown, the power of factory girl can be seen on line 5 where the table that field is dependent upon to be created can simple be defined by refering to the table factory. All model factories were constructed using the same method disaplyed in this example. Once the factories for each model had been defined RSpec test cases were written for requirements 1 to 12 of the functional specification requirements, see Table **??**, to test the functionality of the requirements, If a table can be edited and the edit is saved to the system. Table, Listing 16, and field, Listing 17, are included to show how an example of how all the rspec test cases were coded, the complete code in included in the online upload.

Once the tests had been written they could all be ran at once using the 'rspec

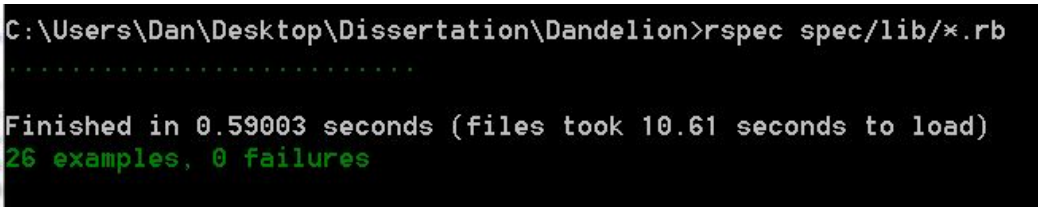

Figure 16: Cmd line ouput running all RSpec test on project

spec/lib/*.rb', the asterix wild card runs all of the tests together. Twenty six test were ran all togther and they all passed, see Figure 16. However this does not completely test the requirement as it is not performed by a user, to complete the testing for the requirement the system was used as a user would peform each action and the server log, see Listing 18, was read and database was queried, see Listing 19 to ensure the each requirements action were carried out. The server log shows the cascade of deletions carried out due to dependencies, the console query confirms the deletion of the table and fields, employee table is no longer in the database and there are no fields in the databse as all fields in the database belonged to the employee table. This method of simulating the user action, observing server logs and querying the database was taken as the manual method for testing.

Requirements 13 to 16 of the functional specification requirements, see Table **??**, were tested manually. An example of how, A Project was navigated to using the website, the tables loaded in the browser, those loaded in the server logs and those queried in the console were all scrutinized to ensure that all tables for that project were loaded and only those tables. Requirement 5 in RSpec test suite **??** was also test using this manner.

Listing 15: Factory for field

```
1  FactoryGirl.define do
2    factory :field do
3      name { "field_" + random_word_lc }
4      description "Dummy_Information"
5      table
6      data_type_id 1
7    end
8  end
9  def random_word_lc
10   ('a'..'z').to_a.shuffle.join
11 end
```

Listing 16: RSpec tests for table

```
1  require "rails_helper"
2
3  describe Table do
4
5    it "can be created" do
6      table = create( :table)
7      expect(Table.where(id: table.id).length).to eq(1)
8    end
9
10   it "can not be created without a project" do
11     table = build( :table, project_id: 1)
12     expect(table).to_not be_valid
13   end
14
15   it "can be edited" do
16     table = create( :table, name:"Table")
17     table.name = "Change"
18     table.save
19     expect(Table.where(id: table.id).first.name).to eq(
         "Change")
20   end
21
22   it "can be deleted" do
23     table = create( :table)
24     table.destroy
25     expect(Table.where(id: table.id).length).to eq(0)
26   end
27
28   it "deletes associated tables when deleted" do
29     table = create( :table)
30     field = create(:field, table_id: table.id)
31     table.destroy
32     expect(Field.where(id: field.id).length).to eq(0)
33   end
34
35 end
```

Listing 17: RSpec tests for field

```
1  require "rails_helper"
```

```
2
3  describe Field do
4
5    it "can be created" do
6      field = create( :field)
7      expect(Field.where(id: field.id).length).to eq(1)
8    end
9
10   it "can not be created without a project" do
11     field = build( :field, table_id: 1)
12     expect(field).to_not be_valid
13   end
14
15   it "can be edited" do
16     field = create( :field, name:"field")
17     field.name = "change"
18     field.save
19     expect(Field.where(id: field.id).first.name).to eq(
         "change")
20   end
21
22   it "can be deleted" do
23     field = create( :field)
24     field.destroy
25     expect(Field.where(id: field.id).length).to eq(0)
26   end
27
28   it "deletes associated fields when deleted" do
29     field = create( :field)
30     validation_assignment = create(:
         validation_assignment, field_id: field.id)
31     field.destroy
32     expect(ValidationAssignment.where(id:
         validation_assignment.id).length).to eq(0)
33   end
34
35   it "can have a given validation" do
36     field = create( :field)
37     validation = create(:validation)
38     validation_assignment = create(:
         validation_assignment, field_id: field.id,
```

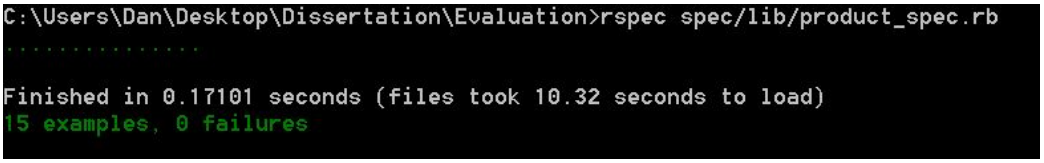

Figure 17: Result of running Product test suite

```
39          validation_id: validation.id)
40        expect(ValidationAssignment.where(id:
              validation_assignment.id).first.field_id).to eq(
              field.id)
41      end
42
43  end
```

see Listing 20 see Figure **??**

Test case requirements, table 5, 1-3,5,7-12 were proved met by the Project suite, Listing 20 being able to run and also demonstrating being able to describe a specfifc failure occuring, see Figure 19. All requirements 2-12 can be observed by inspecting the Project suite, while subjective to the person inspecting there is enough evidence to justify each requirement. For example 4 and 6 require human readable text, the test cases are understandable and the behaviour under test can be inferred by reading the source code of the test cases aswell as when they are ran. 7-12 can be observed by reading the validations under test and calculating if the generated value is correct.

Analysing a test suite, Listing 20, produced for Products, defined in chapter 6, it can be concluded that the test cases are grouped via field and in a logical order, passing requirements 3,4 for Test suite, table 6. Querying the database and cross refrencing the fields and validations belonging to products between those tested in the suite requirement 2 can be confirmed to met aswell. The final requirement, 1, was validated as met by running the test suite, figure 17.

All requirements were concluded to be met, however the majority of these were done manually can can be subjective. The output of the system is fully functional and serves the purpose it intended to.

Table 4: Formal Specification Requirements Testing

ID	Requirement	Written	Manual
1	A user can create a Project	**Y**	**Y**
2	A user can edit a Project	**Y**	**Y**
3	A user can delete a Project, associated Tables are also deleted	**Y**	**Y**
4	A user can create a Table, only for a given Project	**Y**	**Y**
5	A user can edit a Table	**Y**	**Y**
6	A user can delete a Table, associated Fields are also deleted	**Y**	**Y**
7	A user can create a Field, only for a given Table	**Y**	**Y**
8	A user can edit a Field	**Y**	**Y**
9	A user can delete a Field, associated validations are also deleted	**Y**	**Y**
10	A user can assign a validation, only for a given Field	**Y**	**Y**
11	A user can only create a value for a assigned validation	**Y**	**Y**
12	A user can edit a value	**Y**	**Y**
13	A user can view all Projects	**N**	**Y**
14	A user can view all Tables associated to a given Project	**N**	**Y**
15	A user can view all Fields associated to a given Table	**N**	**Y**
16	A user can view all validations and values associated to a given field	**N**	**Y**

Table 5: RSpec Test Case Requirements Testing

ID	Requirement	Written	Manual
1	RSpec test case should be runnable	**N**	**Y**
2	RSpec test case should only test one behaviour	**N**	**Y**
3	RSpec test case must should test behaviour intended	**N**	**Y**
4	RSpec descriptor must be human readable	**N**	**Y**
5	When test case fails, its output must specify exact behaviour at fault	**N**	**Y**
6	RSpec test case should have human readable syntax	**N**	**Y**
7	Be able to generate an Integer that satisfies all validations and their values assigned to a field	**N**	**Y**
8	Be able to generate an Integer that does not satisfy a validation but satisfies all other validations and their values assigned to a field	**N**	**Y**
9	Be able to generate a Float that satisfies all validations and their values assigned to a field	**N**	**Y**
10	Be able to generate a Float that does not satisfy a validation but satisfies all other validations and their values assigned to a field	**N**	**Y**
11	Be able to generate a String that satisfies all validations and their values assigned to a field	**N**	**Y**
12	Be able to generate a String that does not satisfy a validation but satisfies all other validations and their values assigned to a field	**N**	**Y**

Table 6: RSpec Test Suite Requirements Testing

ID	Requirement	Written	Manual
1	RSpec test suite should be runnable	**N**	**Y**
2	RSpec test suite should be contain all test cases for a table	**N**	**Y**
3	RSpec test cases should be grouped via field	**N**	**Y**
4	RSpec test cases should be in a logical order	**N**	**Y**
5	RSpec test suite must be available for download	**N**	**Y**

6 Chapter 6: Results and Discussion

6.1 Evaluation Subject Requirements

For evaluation the following three criteria were desired, all three were not necessary but these were the main factors in choosing subjects. The criteria was chosen to similute useage of the system as close as possible and to provide the best results for analysis.

1. **A Specfication**

   A description of the intended program behaviour that was seperate from the code such as a design specification from which to construct the functional specification using the system. This mimics how a user will use the system.

2. **Database**

   To use a mixture of validations supported and not supported on the data types supported and also have a range of table sizes. Provide a range of results for analysis.

3. **Test cases**

   Have an already established suite of test cases, preferably RSpec to compare with. To compare results and analyse efficency and effectiveness.

[28] [29]

Redmine, Fat Free CRM, Dispora, Amahi and Selfstarter from Derailer[29] and Inoshi, RubyTime, RubyURL and Tracks from Rubicon[28] evaluation subjects were considered but none of them met the desried criteria to a high enough level. Therefore four tables will be constructed to be evaluated. The three tables will range in size, data types used and validations. The specification for the tables to be created are as follows:

Product

1. code : Integer

   Cannot be blank, greater than 0, less than 10000

2. name : String

   Can be blank, format /\A[a-zA-Z]+\z/

3. stock : Integer

   Cannot be blank, greater than or equal to 0, divisible by 5

4. price : Float

   Cannot be blank, greater than 0, less than 10000

Employee

1. Title : String

   Can be blank, Inclusion [Miss, Mrs, Mr, Dr, Prof]

2. first_name : String

   minimum length 5

3. last_name : String

   cannot be blank, maximum length 20

4. position : String

   Cannot be blank, Exclusion [Boss, CEO]

5. age : Integer

   greater than or equal to 0, less than or equal to 100

6. hourly_pay : Float

   greater than or equal to 6.12, exclusion[23.45]

Account

1. account_type : Integer

   Can be blank, divisible by 63

2. name : String

   Can be blank, format /\A[a-zA-Z]+[_][a-zA-Z]+\z/

3. rating : float

   Cannot be blank, less than 10000, divisible by 0.1

Once the specifcations were entered and generated, producing a total of 44 test cases for all three tables . The tests were ran there was one failure. This was the system failing to produce a float that isloated the less than 10000 field on table Account, see Figure 18. However on looking at the test it is

Table 7: Mutation Testing

ID	Change	Detected
1	Divisible by 63 removed from Account, account_type	**Y**
2	Divisible by 63 changed to 79 from Account, account_type	**N**
3	Can be blank changed to can not, from Account, account_type	**Y**
4	Tests 1 and 3 at the same time	**Y**
5	Greater than or equal to 0 removed from Product, stock	**Y**
6	Exclusion removed from Employee, position	**Y**
7	Exclusion and maximum length from Employee, position, last_name	**Y**
8	Format /\A[a-zA-Z]+\z/added to Employee, first_name	**Y**
9	Minimum Length 10 added to Account, name	**N**
10	Uniqueness true added to Product, code	**N**

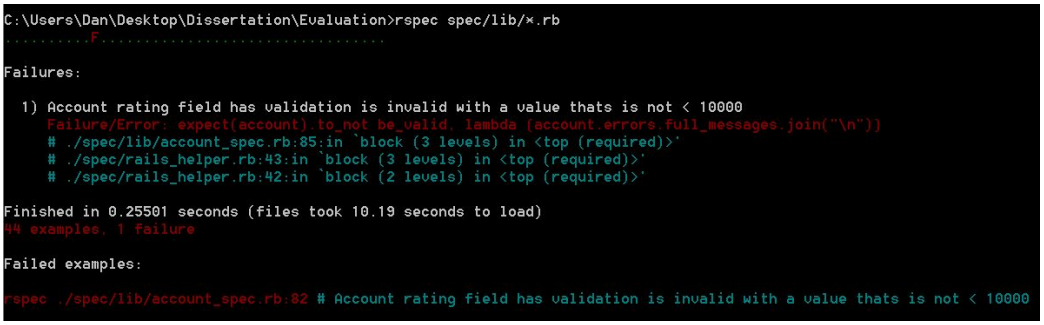

Figure 18: An example of a failing Rspec test case produced by the system

clear the test is at fault and not the users sytem under test with the error message in place of the value, see Figure 19.

For the first part of the evaluation mutation testing will be used, validations will be removed, altered and added and the test suites will be ran, this will demonstrate wether the test suites can pick up the missing validation. A Y in the detected column indicates the tests detected the change, whereas an N indicates they did not.

For the second part of mutation testing the potentail effectiveness of using the system was evaluated.

It took three minutes thirty seconds to construct the formal specifcation for Project, three minute fifty five seconds for Employee and two minutes and

```
it "is invalid with a value thats is not < 10000" do
  account = build(:account, rating: "!!!ERROR GENERATING VALUE")
  if account.respond_to?(:valid?)
    expect(account).to_not be_valid, lambda {account.errors.full_messages.join("\n")}
  end
end
```

Figure 19: An example of a test case where a value could not be generated

sixteen seconds for Account. Giving a total of nine minutes and thrity six seconds to construct the functional specifications on the system. The total test cases produced in all three suites was forty four, with one having not been able to generate a value, so forty three test cases. These forty three test cases were ran and all had the desried behaviour. This gives an average of thirteen seconds to produce a valid working test case, which is a credible speed and is certainly faster than the time it takes to produce a manually written test case.

The test suite and individual test cases, see Figure 20, showed good readability and could be understood to a standard high enough to locate the behaviour at fault. The layout and overall structure is logical and is as expected. Thes factors show that the test suite produced could serve as an initail test file and a foundation for which more complex test are added to. The suite would be useful as the initail tests in a test driven development style of development.

7 Chapter 7: Conclusions

7.1 System

7.1.1 Improvements to current system

Some improvements that could be made to the current system are:

1. Level of user freindliness

   Currently there are no restrictions on what combinations of validations can be chosen, this allows the user to set fields with validations that have no possible valid value. This was an oversight as it was expected a user had a well designed and defined specfication. The current error handling for not generating a value is to return a string that has

high visibility, this was to ensure the test suite was still generated, an improvement could be prompting the user with an error message. Navigating over two layers, e.g from value to table consists of clicking back multiple time or project then table, having the hierachy line that describes the current position as linkable, E.g can click on the table name to go to the table would improve navigation.

2. Border cases for numerical tests

   Border case testing, testing 9, 10, 11 for a maximum value of 10 on a validation is a standard usually followed in automated testing but not in RSpec test cases. This could be introduced as a feature that could be toggled, some users may not want it as it would produce a lot of test cases they may not want.

3. Database fetching efficency

   There a few difffern't ways of fetching data from the database in ruby. One method that isnt implemented fetches all relevant entries and stores them in cache, which can be queried further on, reducing the amount of calls to the database. The differn't methods could be consulted for each fecth to increase optimisation.

7.1.2 System Expandability

Import, Export, Use it

1. more validations
2. relations
3. more data types
4. generating customisabel/ all tables etc etc
5. could use knowledge to produce factory girl skeletons

7.2 Conclusion

This project started out with the motivation of bringing the benefits of automated testing to Ruby on Rails. Through research it was discovered there is automated testing but it does not fit with the Test Driven and iterational

style of testing that Ruby on Rails encourages and is practiced in development of its applications. The aim was refined to creating a system that automatically generates RSpec test cases for model validation in Ruby on Rails Applications. This refined aim fit the natural development cycle of Ruby on Rails applications by automating the writing of test cases that will be written manually by a developer. The test suite generated is basic, that is it only tests for the validations attached to fields. It does however serve as a good foundation from which to add test cases to and as the start of testing in the development cycle.

Listing 18: Server log for deleting Employee table

```
1  #destroy as HTML
2  oken"=>"tJA1Mc52lZYdmAqYnAQqOtpMpZvKp8LS/0
       yVeb4nAmvlj2T26Mo8sHqAFNmc78TPNEwDB0ET4xY5fDeddcrKsg
       ==", "project_id"=>"1",
3
4    "projects".* FROM "projects" WHERE "projects"."id" =
         ? LIMIT 1  [["id", 1]]
5  "tables".* FROM "tables" WHERE "tables"."project_id" =
       ? AND "tables"."id" = ? LIMIT 1  [["project_id", 1],
       ["id", 1]]
6
7  fields".* FROM "fields" WHERE "fields"."table_id" = ?
       [["table_id", 1]]
8  .0ms)  SELECT "validation_assignments".* FROM "
       validation_assignments" WHERE "
       validation_assignments"."field_id" = ?
9
10 "values".* FROM "values" WHERE "values"."
       validation_assignment_id" = ? LIMIT 1  [["
       validation_assignment_id", 3]]
11 lues" WHERE "values"."id" = ?  [["id", 3]]
12 lidation_assignments" WHERE "validation_assignments"."
       id" = ?  [["id", 3]]
13 "values".* FROM "values" WHERE "values"."
       validation_assignment_id" = ? LIMIT 1  [["
       validation_assignment_id", 5]]
14 lues" WHERE "values"."id" = ?  [["id", 5]]
15 lidation_assignments" WHERE "validation_assignments"."
       id" = ?  [["id", 5]]
16 "values".* FROM "values" WHERE "values"."
```

```
   validation_assignment_id" = ? LIMIT 1  [["
   validation_assignment_id", 6]]
17 lues" WHERE "values"."id" = ?  [["id", 6]]
18 lidation_assignments" WHERE "validation_assignments"."
   id" = ?  [["id", 6]]
19 "values".* FROM "values" WHERE "values"."
   validation_assignment_id" = ? LIMIT 1  [["
   validation_assignment_id", 7]]
20 lues" WHERE "values"."id" = ?  [["id", 7]]
21 lidation_assignments" WHERE "validation_assignments"."
   id" = ?  [["id", 7]]
22 elds" WHERE "fields"."id" = ?  [["id", 1]]
23 .0ms)  SELECT "validation_assignments".* FROM "
   validation_assignments" WHERE "
   validation_assignments"."field_id" = ?
24
25 "values".* FROM "values" WHERE "values"."
   validation_assignment_id" = ? LIMIT 1  [["
   validation_assignment_id", 8]]
26 lues" WHERE "values"."id" = ?  [["id", 8]]
27 lidation_assignments" WHERE "validation_assignments"."
   id" = ?  [["id", 8]]
28 "values".* FROM "values" WHERE "values"."
   validation_assignment_id" = ? LIMIT 1  [["
   validation_assignment_id", 9]]
29 lues" WHERE "values"."id" = ?  [["id", 9]]
30 lidation_assignments" WHERE "validation_assignments"."
   id" = ?  [["id", 9]]
31 elds" WHERE "fields"."id" = ?  [["id", 2]]
32 .0ms)  SELECT "validation_assignments".* FROM "
   validation_assignments" WHERE "
   validation_assignments"."field_id" = ?
33
34 "values".* FROM "values" WHERE "values"."
   validation_assignment_id" = ? LIMIT 1  [["
   validation_assignment_id", 1]]
35 lues" WHERE "values"."id" = ?  [["id", 1]]
36 lidation_assignments" WHERE "validation_assignments"."
   id" = ?  [["id", 1]]
37 "values".* FROM "values" WHERE "values"."
   validation_assignment_id" = ? LIMIT 1  [["
```

```
   validation_assignment_id", 2]]
38 lues" WHERE "values"."id" = ?  [["id", 2]]
39 lidation_assignments" WHERE "validation_assignments"."
   id" = ?  [["id", 2]]
40 "values".* FROM "values" WHERE "values"."
   validation_assignment_id" = ? LIMIT 1  [["
   validation_assignment_id", 10]]
41 lues" WHERE "values"."id" = ?  [["id", 10]]
42 lidation_assignments" WHERE "validation_assignments"."
   id" = ?  [["id", 10]]
43 "values".* FROM "values" WHERE "values"."
   validation_assignment_id" = ? LIMIT 1  [["
   validation_assignment_id", 11]]
44 lues" WHERE "values"."id" = ?  [["id", 11]]
45 lidation_assignments" WHERE "validation_assignments"."
   id" = ?  [["id", 11]]
46 "values".* FROM "values" WHERE "values"."
   validation_assignment_id" = ? LIMIT 1  [["
   validation_assignment_id", 12]]
47 lues" WHERE "values"."id" = ?  [["id", 12]]
48 lidation_assignments" WHERE "validation_assignments"."
   id" = ?  [["id", 12]]
49 elds" WHERE "fields"."id" = ?  [["id", 3]]
50 .0ms)  SELECT "validation_assignments".* FROM "
   validation_assignments" WHERE "
   validation_assignments"."field_id" = ?
51
52 "values".* FROM "values" WHERE "values"."
   validation_assignment_id" = ? LIMIT 1  [["
   validation_assignment_id", 13]]
53 lues" WHERE "values"."id" = ?  [["id", 13]]
54 lidation_assignments" WHERE "validation_assignments"."
   id" = ?  [["id", 13]]
55 "values".* FROM "values" WHERE "values"."
   validation_assignment_id" = ? LIMIT 1  [["
   validation_assignment_id", 14]]
56 lues" WHERE "values"."id" = ?  [["id", 14]]
57 lidation_assignments" WHERE "validation_assignments"."
   id" = ?  [["id", 14]]
58 "values".* FROM "values" WHERE "values"."
   validation_assignment_id" = ? LIMIT 1  [["
```

```
   validation_assignment_id", 15]]
59 lues" WHERE "values"."id" = ?  [["id", 15]]
60 lidation_assignments" WHERE "validation_assignments"."
   id" = ?  [["id", 15]]
61 elds" WHERE "fields"."id" = ?  [["id", 4]]
62 bles" WHERE "tables"."id" = ?  [["id", 1]]
63
64 :3000/projects/1
65 ctiveRecord: 13.0ms)
```

Listing 19: Rails console query for table and field existance

```
1  ::1 at 2017-05-03 01:17:12 +0100
2  #index as HTML
3  1"}
4  "projects".* FROM "projects" WHERE "projects"."id" =
   ? LIMIT 1  [["id", 1]]
5  tables".* FROM "tables" WHERE "tables"."project_id" = ?
       [["project_id", 1]]
6  rb within layouts/application (3.0ms)
7  s: 63.0ms | ActiveRecord: 0.0ms)
8
9  irb(main):011:0> Table.all
10  Table Load (1.0ms)  SELECT "tables".* FROM "tables"
          => #<ActiveRecord::Relation [#<Table id: 2,
     name: "Store", description: "Store Information",
     project_id: 1, created_at: "2017-04-24 22:17:38",
     updated_at: "2017-04-24 22:17:38">, #<Table id: 3,
     name: "Vehicle", description: "Vehicle Information
     ", project_id: 1, created_at: "2017-04-24
     22:17:38", updated_at: "2017-04-24 22:17:38">]>
11 irb(main):012:0> Field.all
12  Field Load (0.0ms)  SELECT "fields".* FROM "fields" =>
      #<ActiveRecord::Relation []>
```

Listing 20: Test suite output for Product defined in chapter 6

```
1 require "rails_helper"
2
3 describe Product do
4
5         it "has a valid factory" do
```

```
6               expect(build(:product)).to be_valid
7           end
8
9           describe "code field has validation" do
10
11              it "can not be blank" do
12                  product = build(:product, code:
                        nil)
13                  if product.respond_to?(:valid?)
14                      expect(product).to_not
                            be_valid, lambda {
                            product.errors.
                            full_messages.join("
                            \n")}
15                  end
16              end
17
18              it "is valid with a generated value" do
19                  product = build(:product, code:
                        9314)
20                  if product.respond_to?(:valid?)
21                      expect(product).to
                            be_valid, lambda {
                            product.errors.
                            full_messages.join("
                            \n")}
22                  end
23              end
24
25              it "is invalid with a value thats is
                    not > 0" do
26                  product = build(:product, code:
                        -487078)
27                  if product.respond_to?(:valid?)
28                      expect(product).to_not
                            be_valid, lambda {
                            product.errors.
                            full_messages.join("
                            \n")}
29                  end
30              end
```

```
31
32                  it "is invalid with a value thats is
                        not < 10000" do
33                      product = build(:product, code:
                            449752)
34                      if product.respond_to?(:valid?)
35                          expect(product).to_not
                                be_valid, lambda {
                                product.errors.
                                full_messages.join("
                                \n")}
36                      end
37                  end
38
39              end
40
41              describe "name field has validation" do
42
43                  it "can be blank" do
44                      product = build(:product, name:
                            nil)
45                      if product.respond_to?(:valid?)
46                          expect(product).to
                                be_valid, lambda {
                                product.errors.
                                full_messages.join("
                                \n")}
47                      end
48                  end
49
50                  it "is valid with a generated value" do
51                      product = build(:product, name:
                            "XoJbfmoPkAHFv")
52                      if product.respond_to?(:valid?)
53                          expect(product).to
                                be_valid, lambda {
                                product.errors.
                                full_messages.join("
                                \n")}
54                      end
55                  end
```

```
56
57                  it "is invalid with a value thats is
                        not regex /\A[a-zA-Z]+\z/" do
58                          product = build(:product, name:
                                "CaQEWaIldTMdNiNN!'}??-@)
                                &\"(-'*:]!'(
                                FnahDn8DrLNY6_5u04e0jrEPQJmWX
                                ")
59                          if product.respond_to?(:valid?)
60                                  expect(product).to_not
                                        be_valid, lambda {
                                        product.errors.
                                        full_messages.join("
                                        \n")}
61                          end
62                  end
63
64          end
65
66          describe "stock field has validation" do
67
68                  it "can not be blank" do
69                          product = build(:product, stock
                                : nil)
70                          if product.respond_to?(:valid?)
71                                  expect(product).to_not
                                        be_valid, lambda {
                                        product.errors.
                                        full_messages.join("
                                        \n")}
72                          end
73                  end
74
75                  it "is valid with a generated value" do
76                          product = build(:product, stock
                                : 902865)
77                          if product.respond_to?(:valid?)
78                                  expect(product).to
                                        be_valid, lambda {
                                        product.errors.
                                        full_messages.join("
```

```
                                        \n")}
79                      end
80              end
81
82              it "is invalid with a value thats is
                    not >= 0" do
83                      product = build(:product, stock
                            : -140355)
84                      if product.respond_to?(:valid?)
85                              expect(product).to_not
                                    be_valid, lambda {
                                    product.errors.
                                    full_messages.join("
                                    \n")}
86                      end
87              end
88
89              it "is invalid with a value thats is
                    not divisible 5" do
90                      product = build(:product, stock
                            : 587473)
91                      if product.respond_to?(:valid?)
92                              expect(product).to_not
                                    be_valid, lambda {
                                    product.errors.
                                    full_messages.join("
                                    \n")}
93                      end
94              end
95
96      end
97
98      describe "price field has validation" do
99
100             it "is valid with a generated value" do
101                     product = build(:product, price
                            : 5563.56)
102                     if product.respond_to?(:valid?)
103                             expect(product).to
                                    be_valid, lambda {
                                    product.errors.
```

```
                                        full_messages.join("
                                        \n")}
104                             end
105                     end
106
107                     it "is invalid with a value thats is
                            not > 0" do
108                             product = build(:product, price
                                    : -67150.92)
109                             if product.respond_to?(:valid?)
110                                     expect(product).to_not
                                            be_valid, lambda {
                                            product.errors.
                                            full_messages.join("
                                            \n")}
111                             end
112                     end
113
114                     it "is invalid with a value thats is
                            not < 10000" do
115                             product = build(:product, price
                                    : 59127.4)
116                             if product.respond_to?(:valid?)
117                                     expect(product).to_not
                                            be_valid, lambda {
                                            product.errors.
                                            full_messages.join("
                                            \n")}
118                             end
119                     end
120
121             end
122
123 end
```

References

[1] C. Barker, "The top 10 it disasters of all time," 2007.

[2] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[3] J. Gliek, "A bug and a crash," 1996.

[4] P. A. McQuaid, "Software disastersunderstanding the past, to improve the future," *Journal of Software: Evolution and Process*, vol. 24, no. 5, pp. 459–470, 2012.

[5] S. Nidhra and J. Dondeti, "Blackbox and whitebox testing techniques-a literature review," *International Journal of Embedded Systems and Applications (IJESA)*, vol. 2, no. 2, pp. 29–50, 2012.

[6] M. Young, *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley India Pvt. Limited, 2008.

[7] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[8] R. on Rails, "A guide to testing rails applications," 2017.

[9] Django, "Django faq," 2017.

[10] SHUUP, "25 of the most popular python and django websites," 2015.

[11] Alexa, "The top 500 websites," 2017.

[12] Coderfactory, "Top 15 sites built with ruby on rails," 2017.

[13] Infoq, "Facebook: Mvc does not scale, use flux instead," 2014.

[14] Ruby, "About ruby," 2016.

[15] R. on Rails, "Activerecord::connectionadapters::schemastatements," 2017.

[16] M. Maguire, "Methods to support human-centred design," *International journal of human-computer studies*, vol. 55, no. 4, pp. 587–634, 2001.

[17] C. Abras, D. Maloney-Krichmar, and J. Preece, "User-centered design," *Bainbridge, W. Encyclopedia of Human-Computer Interaction. Thousand Oaks: Sage Publications*, vol. 37, no. 4, pp. 445–456, 2004.

[18] G. Hub, "Git hub," 2017.

[19] Airbnb, "Ruby style guide," 2017.

[20] Rubular, "Rubular a ruby regular expression editor," 2017.

[21] C. Tran, "Annotate models," 2017.

[22] R. on Rails, "http://guides.rubyonrails.org/association_basics.html," 2017.

[23] Bootstrap, "Bootstrap," 2017.

[24] R. on Rails, "Rails routing from the outside in," 2017.

[25] T. Lord, "regexp-examples," 2017.

[26] S. Abshere, "to_regexp," 2017.

[27] thoughtbot, "Factory giel," 2017.

[28] J. P. Near and D. Jackson, "Rubicon: bounded verification of web applications," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, p. 60, ACM, 2012.

[29] J. P. Near and D. Jackson, "Derailer: interactive security analysis for web applications," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 587–598, ACM, 2014.