FormsFX

Project Report

Sacha Schmid
Rinesch Murugathas

Dirk Lemmermann
Dieter Holz

Windisch, 18 August 2017

1. Summary

This report describes the project «FormsFX» and contains information about architectural decisions, technological specifications, and challenges faced during development. It also compares different solutions for the challenges in this project and how they compare to each other.

Forms are an essential part of many business applications. In JavaFX, creating such forms can be a tedious and time-consuming process with little to no pre-defined components for things like validation.

The result of this project is a JavaFX framework that helps developers create powerful forms with little manual effort. Instead of spending time on tedious manual work, developers can instead focus on the semantic specifics of their forms and how the data entry could be handled most efficiently.

1.1. Fact Sheet

Project Name	FormsFX
Project Duration	20 February 2017 – 18 August 2017
Team Members	Sacha Schmid, Rinesch Murugathas
Customer	Dirk Lemmermann Software & Consulting
Coach	Dieter Holz
Repository	https://github.com/DieterHolz/FormsFX

Table 1 Fact sheet about the 'FormsFX' project

Table of Contents

2. Introduction

This report details the processes and findings of the IP6 project «FormsFX» along with architectural details, design decisions, and analysis on relevant topics and competitors.

Most business applications use forms in some way or another to collect and manipulate data. Some examples for this might be gathering customer addresses or credit card information.

In JavaFX there is not a lot of support for such forms. There are individual components to handle user interaction. This, however, does not include bindings to a model, support for multi-language applications, or validation methods.

Creating forms in JavaFX is a complicated process that requires a lot of manual effort and eventually leads to loads of repeated code and, thus, a heavily error-prone process. FormsFX is a framework that allows developers to create complex forms in an easy manner by hiding much of the complexity and by removing the need for tedious, repetitive, manual work.

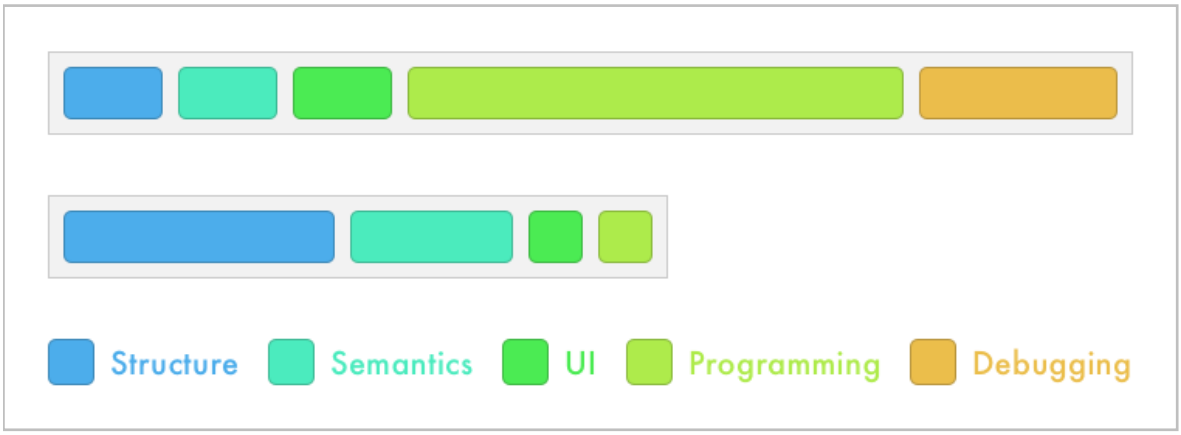

Figure 1 Distribution of work with JavaFX forms on top and FormsFX below

FormsFX not only saves a developer's time, but also allows them to spend their precious time on the things that actually matter, like structure and semantics.

First and foremost, FormsFX is a tool to increase *developer productivity*. Developers should not be forced to waste hours upon hours writing the same functionality, but instead be able to focus on how the form should be structured and which information is essential. In the end, this also helps end users as they get a more usable form.

The main part of this report documents the individual pieces of the FormsFX tool and how they interact and build upon each other.

Throughout this report, the term «user» will be in reference to developers who use the FormsFX framework. Users of the resulting forms will be referred to as «end users».

3. Previous Situation

3.1. Problems

Java as a language is often criticised for its boilerplate and often repetitive code. Of course this also applies to creating forms, where the problem is especially obvious. The process involves various bindings and listeners that have to be created manually. Often, one of these bindings can be forgotten or only applied in one direction, which then breaks the form in a rather non-obvious way. This leads to time spent on unnecessary debugging and lots of frustration on the developer's side.

The following listing shows a snippet from a form developed in JavaFX. It is clearly visible that even a simple form requires lots of manual bindings among other complexities.

```
Bindings.bindBidirectional(tfName.textProperty(),
country.nameProperty());
Bindings.bindBidirectional(tfPopulation.textProperty(),
country.populationProperty(), new NumberStringConverter());
Bindings.bindBidirectional(tfCountryCode.textProperty(),
country.countryCodeProperty());
country.latitudeProperty(), new NumberStringConverter());
Bindings.bindBidirectional(tfLongitude.textProperty(),
country.longitudeProperty(), new NumberStringConverter());
Bindings.bindBidirectional(title.textProperty(),
country.nameProperty());
```

Listing 1 Manual creation of bindings

3.2. Competitors

This project is not the first that tries to solve this problem. There have been other projects, which solve it in their own ways. Two such products we have found are FXForm2 and TornadoFX, which both attempt to make it easier to create forms.

3.2.1. FXForm2

FXForm2, like this project, is also solved in JavaFX and is very similar in the features it offers. The approach with this competitor is that they have solved it like a normal Java application extension, meaning you have to write code like in any other Java application.

This competitor relies heavily on properties and Java beans, from which most aspects of the form are derived. This, however, still involves lots of manual coding, as the bean classes still have to be created manually. One of their tag lines is «don't waste time coding forms, focus on styling,» which is indicative of their priorities. Indeed, the form is created based on the model bean, at which point the user is practically forced to add in custom styling in order to achieve a uniform and visually pleasing form.

Our approach is different. From the beginning, we set out to create a fluent API, which is responsible for creating the form. This is not the case with FXForm2. In FormsFX, as well as in FXForm2, the model has to be created first. However thanks to the fluent API it is much more intuitive and less complicated to create the form itself.

Another big difference is that we have validators directly embedded in our fluent API, whereas with FXForm2 the validators can be annotated in the object class. Also our product offers more validators where the competition offers only a limited amount of validators.

Also, FXForm2 offers the same styling possibilities as our product, however their style is somewhat outdated. That, however, is no problem since the user can style it however they like. The big advantage in FormsFX is that we offer a default styling for the form, so that the form looks uniform and the user does not have to do more if they are satisfied with the default styling.

Further, they offer only the possibility to add all the contents of the form in one column, with the exception that the user can create a skin for the form to do the layout. We offer the possibility to do that directly in our fluent API with the `.span()` option, which defines how many columns a field can take. This allows the user to layout his form however they like.

Another big advantage our product has is the use of custom controls to render fields, whereas in FXForm2, they let the user create the fields first and then add it to the form. With our approach, we relieve the user from creating more tedious boilerplate code as our custom controls are specified for each type. For example, the user wants a field «*name*». Now all they have to do is `Field.ofStringType("name")` and this will create a custom control with a `Label` and `TextField`.

However, one feature they offer and we do not is the possibility to reorder fields at runtime. This is not possible with our product, the user has to define the order of the fields from the beginning. This can easily be adjusted with copy and paste to the desired order.

Lastly, our product has the advantage of handling state changes, which is only possible to a limited degree in FXForm2. This means that it is easily visible when something on the form changes and this is supported by appropriate state stylings.

```
public class SimpleForm {
    @Override
    public String getSampleName() {
        return "Basic form";
    }

    @Accessor(value = Accessor.AccessType.FIELD)
    public class User {
        public StringProperty username = new
SimpleStringProperty();
        public StringProperty password = new
SimpleStringProperty();
    }

    @Override
    public Node getPanel(Stage stage) {
        Pane root = new Pane();

        FXForm form = new FXFormBuilder<>()
                .includeAndReorder("username", "password")
                .build();

        User user = new User();
        form.setSource(user);

        root.getChildren().add(form);
        return root;
    }
}
```

Listing 2 Simple form created with FXForm2

When comparing the above example with the one below, it is visible that the latter has cleaner code. This also leads to the following example being much easier to understand for a new user using FormsFX. Another advantage of FormsFX is the fluent API, which allows the user to easily customise the field further, like setting a label for the field with the `.label()` option and even more (see *6.4.*) With FXForm2 this is not that easily possible and also harder to understand.

```
public class User {
        public StringProperty username = new
SimpleStringProperty();
        public StringProperty password = new
SimpleStringProperty();
}

User u = new User();

Form loginForm = Form.of(
        Group.of(
                Field.ofStringType(u.username)
                        .label("username"),
                Field.ofStringType(u.password)
                        .label("password")
        )
).title("Basic form”);

Pane root = new Pane();

root.getChildren().add(new FormRenderer(loginForm));
return root;
```

Listing 3 Simple form created with FormsFX

3.2.2. TornadoFX

TornadoFX is more than just a framework to build forms. It is intended to create JavaFX applications easier and minimise the amount of code needed to do so. TornadoFX is written in Kotlin and serves as a framework for JavaFX. It has support for many of Kotlin's practical features as well as working with existing JavaFX libraries, like ControlsFX. Therefore, the main problem this tries to solve is not exactly the same as FormsFX.

Even though this framework is not aimed directly at creating forms, it still does a good job at it. It reduces the amount of code needed to build a form and has handy features like checking for changes before saving the form and disabling buttons accordingly. It also has validation features like, required check and showing errors in a `Tooltip`. This is all very similar to our framework, with the difference being that we have a very strong API which does all of that for us in a very simple and easily understandable way. So the

only difference, at first glance, might seem that their framework is written in Kotlin and ours is in JavaFX.

However one of the big difference is that it still requires the user to create the fields beforehand and then bind them with the appropriate property type. There are controls to solve this issue, but there are only a few standard controls available and most of them are application specific and not really created to work within a form. They, however, have a guide, which shows how to create a control if needed, whereas our framework already delivers a simple control for almost every type that can be included in a form. The added bonus of our product is that our form can be extended with custom built controls, whenever more functionality is needed.

This example shows how a simple login form can be created with TornadoFX and it includes also some logic for the buttons.

```
class EnabledView : View("Basic form") {
    val model = ViewModel()
    val username = model.bind { SimpleStringProperty() }
    val password = model.bind { SimpleStringProperty() }

    override val root = form {
        fieldset {
            field("username") {
                textfield(username).required()
            }
            field("password") {
                passwordfield(password).required()
            }
            buttonbar {
                button("Cancel",
ButtonBar.ButtonData.CANCEL_CLOSE).setOnAction {
                    println("do nothing")
                }
                button("OK", ButtonBar.ButtonData.OK_DONE) {
                    enableWhen { model.valid }
                    setOnAction {
                        println("do something")
                    }
                }
            }
        }
        model.validate(decorateErrors = false)
    }
}
```

Listing 4 Simple form created with TornadoFX in Kotlin

With FormsFX the same form can be achieved like follows, however the buttons have to be created outside of the fluent API. With these examples it is now visible that the TornadoFX example above is more error prone than the lower FormsFX one. This is due to the fact that the fields have to be bound by the user, whereas with FormsFX this is done automatically. Another advantage our product brings is that the user can set the required message himself, making it easier for him to configure that instead of having a default message.

```
public class User {
        public StringProperty username = new
SimpleStringProperty();
        public StringProperty password = new
SimpleStringProperty();
}

User u = new User();

Form loginForm = Form.of(
        Group.of(
                Field.ofStringType(u.username)
                        .label("username"),
                Field.ofStringType(u.password)
                        .label("password")
        )
).title("Basic form”);

Pane root = new Pane();

root.getChildren().add(new FormRenderer(loginForm));
return root;
```

Listing 5 Simple form created with FormsFX in JavaFX

4. Architecture

In general, the architecture is made up of two main layers. On one hand, there is a model layer, which defines the semantical relation between all components and processes the form data. On the other hand, there is the view layer, which presents the modelled data to the end users and allows them to interact with the data in order to create and modify it.

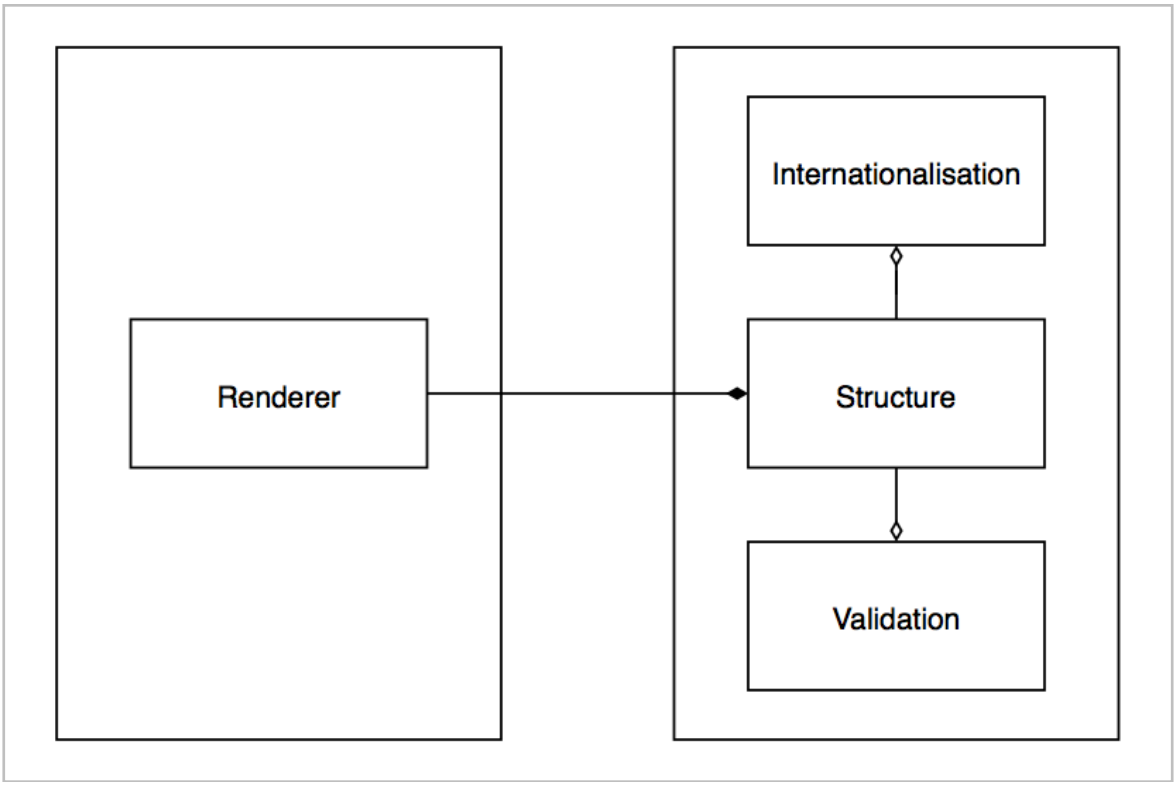

Figure 2 High-level diagram

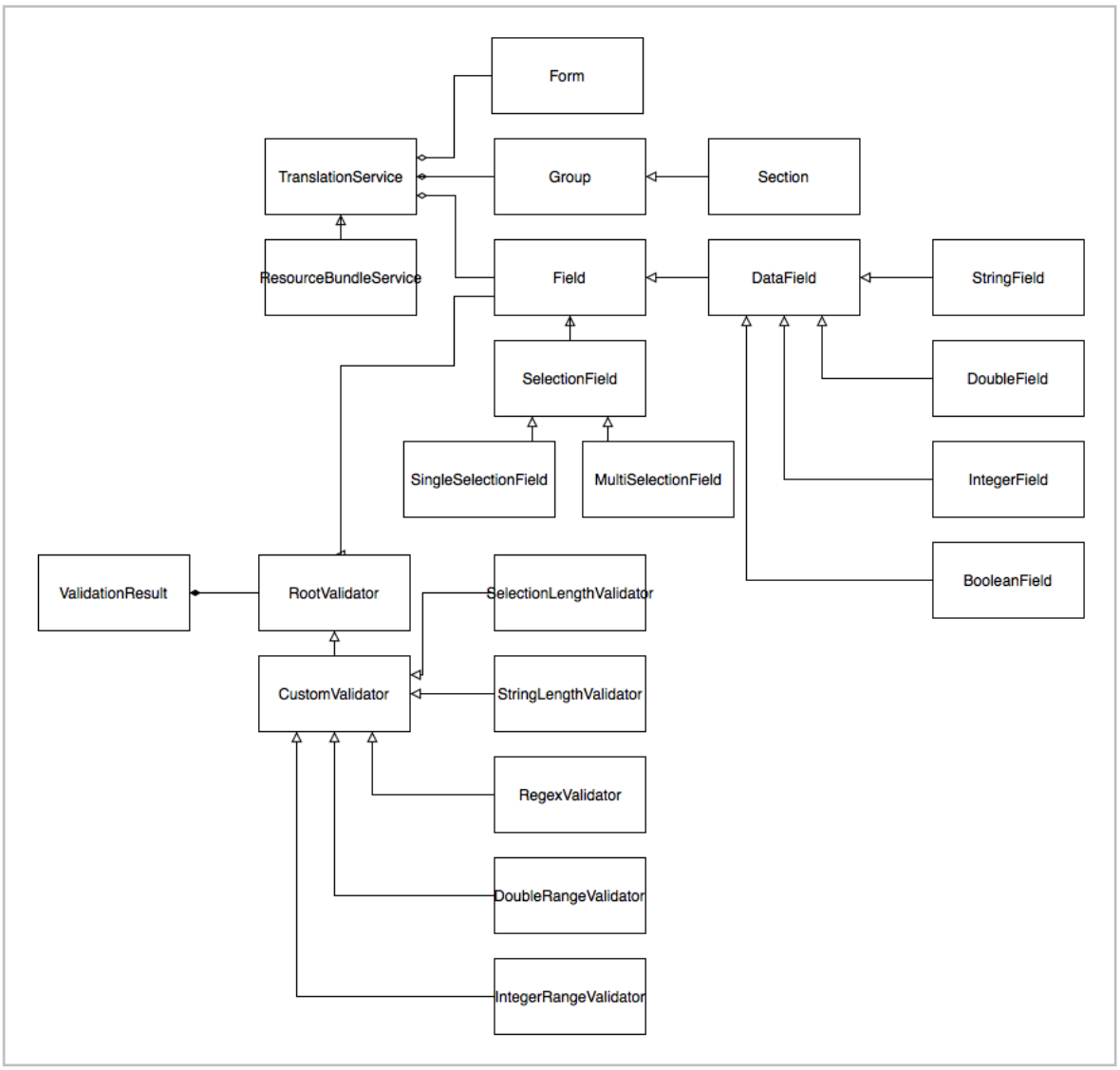

Figure 3 Model layer class diagram

The model layer is made up of structural elements that group, contain, and process form data. This layer is also responsible for validation using a set of pre-defined and custom validators, as well as handling different localisation options.

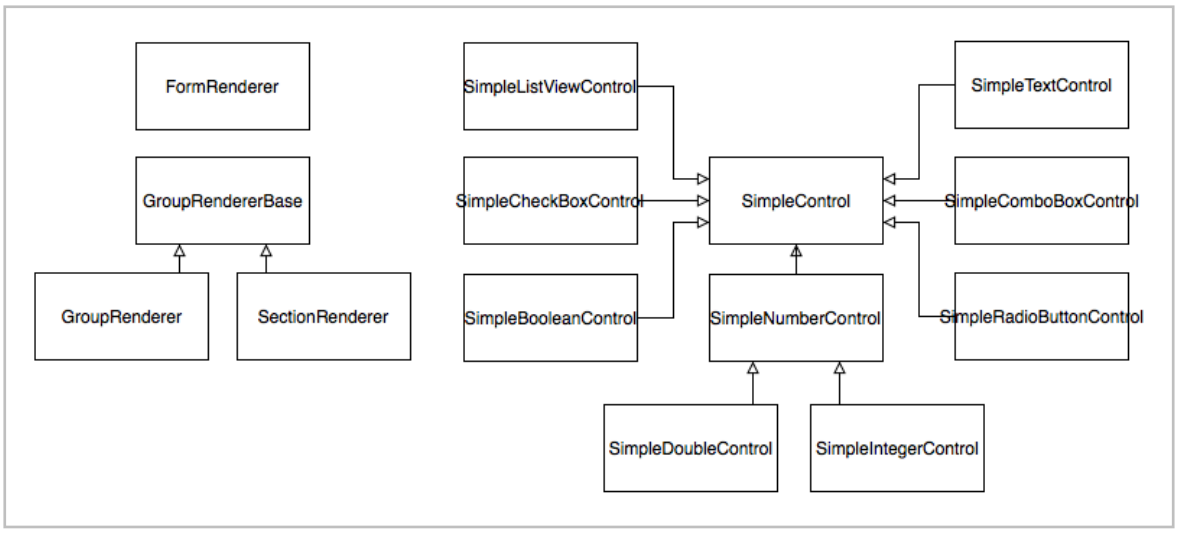

Figure 4 View layer class diagram

The view layer is made up of renderers. These take a structural element from the model layer and turn them into a displayable element that can be included in a GUI application.

The separation between model and view is clear. Both layers serve their own task and rely on each other to fulfil the tasks outside of their own responsibility. This separation of concerns makes the framework much more understandable and allows for more extensibility. This way, especially on the view layer, users can extend existing functionality and provide new interaction models to their own end users, without losing the many benefits of FormsFX.

This extensibility and potential for customisation is a clear goal for FormsFX. Reducing complexity for the expected use cases of a form goes a long way towards higher developer efficiency, however, also providing extension points and paving the way for custom implementations makes FormsFX the ideal tool for forms-based applications.

5. API Design

5.1. Code Versus DSL

With this project there was the possibility to go in different ways with the API design. One of this discussion was to create the API in code versus using a DSL.

This project solved this problem in code for the following reasons. With the code approach there is the advantage of having code completion in the IDE. Lots of developers rely heavily on code completion as it makes it really easy to code and lets the developer be more efficient.

Along with the ease and comfort of code completion comes greater transparency of the API. Developers can often simply start typing a method name and have code completion offer them a list of available methods and their expected arguments and types.

Another advantage this approach gives us, is that there is the possibility of validation. Some form of basic validation is already available in every language and the special advantage is that they can be enhanced further, meaning they can be custom built, which is not possible with DSL.

5.1.1. Kotlin

To create the API design, there was also the possibility to do it in Kotlin (https://goo.gl/rjQvjf). Kotlin is very similar to Java and it features new and additional functionality, since it is more modern than Java.

Kotlin features a somewhat less verbose syntax, at least compared to Java, while inferring a lot of information from the code and its context. This has a variety of benefits as it could potentially streamline some of the previously mentioned tedious handiwork by wrapping it in a more concise and clear syntax, but the approach would still suffer from the repetition. Thus, Kotlin alone would not be the desired silver bullet.

```
// Java

public abstract class Car {
    private String name;

    Car(String name){
        this.name = name;
    }
}

public class Ford extends Car{
    private String model;
    private String color;

    Ford(String model, String color){
        super("Ford");
        this.model = model;
        this.color = color;
    }
}

Ford focus = new Ford("Focus", “blue”);

// Kotlin

abstract class Car(val name: String)

class Ford(val model: String, val color: String) : Car(“Ford")

val focus = Ford("Focus", "Blue")
```

Listing 6 Java vs Kotlin code comparison

Another reason to choose JavaFX over Kotlin is that Kotlin 1.0 was only released a few days before this project started, thus it was fairly new and a potential risk to take. There was not much documentation available at that time and all the problems had to be solved through StackOverflow. Therefore, we decided to stick with JavaFX, since we already know that and were confident in our skills.

5.1.2. YAML

During development there was the possibility to design the API with YAML (*YAML Ain't Markup Language*). This approach has the advantage that it is very easy to understand since the syntax is created to be very much human-readable.

The big advantage of YAML is that it was built from ground up to be simple to use and therefore it allows you to do powerful configuration without having to learn complex concepts.

Another benefit of YAML is that it relies on indentation to derive hierarchy. This would make something like a form with its sub-elements and structure instantly transparent.

```
receipt:     Oz-Ware Purchase Invoice
date:        2012-08-06
customer:
    first_name:   Dorothy
    family_name:  Gale

items:
    - part_no:   A4786
      descrip:   Bucket
      price:     1.47
      quantity:  4

bill-to:  &id001
    street: |
            123 Tornado Alley
            Suite 16
    city:   East Centerville
    state:  KS

ship-to:  *id001
```

Listing 7 YAML sample code

The most obvious drawback is lack of code completion. Personal experience has shown that configuration files are often created by copying and pasting sections of the file and replacing the required values. Having the API in code

makes it easier to write the entire structure by hand and to have a better understanding of the inner workings of what's happening.

In addition to the issue of code completion, having the configuration in another language would also introduce another layer of complexity as the file would first have to be parsed and interpreted. For the user, this would also mean that they have to shift their model of thinking from writing code to writing configuration, which often breaks developer flow.

5.2. Creational Patterns

5.2.1. Factory Pattern

The factory pattern is used to create instances of the structural objects, i.e. Forms, Groups, and Fields. All these classes act as abstract factories that expose methods starting with `of`, e.g. `Form#of`, `Group#of`, and `Field#ofStringType`.

These factories make the structural options for creating a form very transparent. Especially the `ofType` distinction on Fields gives users a clear understanding of the semantics contained within a field.

```
Form.of(
    Group.of(
        Field.ofStringType("Test"),
        Field.ofSingleSelectionType(new ArrayList<>() { … }, 1)
    ),
    Section.of(
        Field.ofDoubleType(3.5),
        Field.ofBooleanType(false)
    )
)
```

Listing 8 Form structure made up using the structural elements

5.2.2. Builder Pattern

FormsFX relies heavily on a simplified form of the builder pattern. This creational pattern offers users a flexible approach to creating and augmenting objects using a *fluent API*.

Unlike the factory pattern and traditional constructors, a builder does not necessarily restrict available options and the order in which they are defined. Instead, users can chain method calls and build up objects or even augment them later on using an API that is based on natural language, i.e. the method calls read fluently.

The advantages of this approach are clear. Users have a way to rapidly build up a form without creating auxiliary variables as they can chain method calls. This makes the form's structure much clearer and groups relevant objects together.

6. Model

6.1. Structure

The model layer relies on a clear structure to build an understandable and usable form. This structure needs to be represented by a transparent API and clear naming conventions.

The structural components each represent their own purpose and delegate relevant work to the lower levels. Like this, a clear separation of concern can be achieved, which in turn keeps the form efficient and extensible.

Semantic issues are a clear focus on this layer. Components, properties and their modification methods, as well as state indicators are named to represent their functionality in a very transparent fashion.

6.2. Form

Forms are the highest semantic level on the structural layer. In general, end users expect to handle exactly one form at a time. For a good user experience, they expect this form to be structured clearly with groupings for connected fields.

In its essence, a form is simply a container for groups and sections, and, by connection, for fields. This brings with it a lot of responsibility, as forms handle some properties that are to apply to all child elements, such as localisation functionality.

The form is then the primary point of interaction for the user. In most cases, checking the state of individual components is not necessary, but instead one only wants to check if the entire form is valid or changed. In this situation, the form acts as a proxy, checking all its children for their state and revealing this aggregate to the user.

As mentioned above, the form is also responsible for handling internationalisation. After localising all the form properties, the localisation service is then passed down to the child elements, where the responsibility for translation is delegated to the other components.

Property	Purpose	Methods	I18N
title	Sets the form title. In forms-centered applications, the window title can be bound to this property.	title(String) getTitle() titleProperty()	x
valid	Determines whether all child elements are currently valid.	isValid() validProperty()	
changed	Determines whether any child elements have changes from their persisted state.	hasChanged() changedProperty()	
persistable	Determines whether all child elements can be persisted. For this, the form needs to have changes, all of which are valid.	isPersistable() persistableProperty()	
translationService	This service is used to translate all translatable values.	i18n(TranslationService) isI18N()	

Table 2 Form properties

6.2.1. Value Handling

Form fields can be bound to external properties. For this, there exist different modes. One mode is to update the model only when the form is persisted, while the other mode does it continuously. These can be changed on the form-level. Internally, the continuous binding mode persists form data upon all valid changes.

6.3. Group and Section

Groups and sections are intermediary component on the structural layer. Just like the form, they act as a sort of wrapper for child components, in this case fields.

As the name implies, these components create a structural grouping or a semantic sectioning of content. The difference between the two components is just that. Groups are a loose coupling of fields where there is no need for a semantic relevance between each field, but instead only a need for structur-

al organisation. Sections, on the other hand, represent both a structural and semantic grouping so they serve a more strict purpose than groups. Sections can also carry a title to identify the purpose of the fields, as well as offering the end user the possibility of collapsing the entire section.

Both groups and sections are essentially collections of fields. Just like the forms delegate work to groups, groups can then delegate work to their fields in a very similar manner.

Groups and sections both offer the possibility to check the changed and valid states, however, this is usually handled through the form itself (see *6.2*.)

Property	Purpose	Methods	I18N
title	Sets the section title. Does not apply to groups.	title(String) getTitle() titleProperty()	x
collapsed	Keeps track of whether the section is currently collapsed. Does not apply to groups.	isCollapsed() collapsedProperty()	
valid	Determines whether all child elements are currently valid.	isValid() validProperty()	
changed	Determines whether any child elements have changes from their persisted state.	hasChanged() changedProperty()	

Table 3 Group and Section properties

6.4. Field

Fields are the smallest unit in a form. They contain and manage the actual values and handle user interaction and input, thus handling the most essential task in the entire form. On a semantic and functional level, fields need to be differentiated based on what types of values they are to contain.

In FormsFX, there is a distinction between two general types of fields. On one hand, there is the `DataField`, a type of field that handles free-form user entry, usually text-based. End users can thus enter any kind of text into these fields. The user input is then converted to an appropriate type that the field can hold as the current value.

On the other hand, there is the SelectionField. These fields offer end users a list of possible items, from which one or more can be selected. End users do not have the option to modify this list in any way other than changing the current selection.

In the context of a form, fields cannot stand on their own. They are to be contained in a group or a section, which in turn is contained in a form. This allows delegation of things like internationalisation to a higher level. The fields can then focus on their own properties.

Property	Purpose	Methods	I18N
label	Describes the field's content in a concise manner. This description is always visible and usually placed next to the editable control.	label(String) getLabel() labelProperty()	x
tooltip	This contextual hint further describes the field. It is usually displayed on hover or focus.	tooltip(String) getTooltip() tooltipPorperty()	x
placeholder	This hint describes the expected input as long as the field is empty.	placeholder(String) getPlaceholder() placeholderProperty()	x
required	Determines, whether entry in this field is required for the correctness of the form.	required(boolean) required(String) isRequired() requiredProperty()	x
editable	Determines, whether end users can edit the contents of this field.	editable(boolean) isEditable() editableProperty()	
valid	Determines, whether the current user input is considered valid.	isValid() validProperty()	
changed	Determines, whether the current user input is different from the persisted value.	hasChanged() changedProperty()	
id	Describes the field with a unique ID. This is not visible directly, but can be used for styling purposes.	id(String) getID() idProperty()	
styleClass	Adds styling hooks to the field. This can be used on the view layer.	styleClass(List<String>) getStyleClass() styleClassProperty()	
span	Determines, how many columns the field should span on the view layer. Can be a number between 1 and 12 or a ColSpan fraction.	span(int) span(ColSpan) getSpan() spanProperty()	
errorMessages	Holds a list of all current error messages that validators have raised. Usually displayed in a tooltip.	getErrorMessages() errorMessagesProperty()	x
renderer	Determines the control that is used to render this field on the view layer.	render(SimpleControl) getRenderer()	

Table 4 Field properties

6.4.1. Value Handling

Fields handle values in multiple levels. The first level is the `user input` property. This value is directly modified by users and not validated, i.e. it is an exact representation of what the user entered. If the validation passes,

the user input is then transformed and stored in the `value` property. The value property can then be persisted using the `persist()` method, which not only updates the `persistent value` property, but also updates any bindings to model classes. Alternatively, fields can be reset to their persisted value using the `reset()` method.

6.4.2. DataField

A `DataField` contains arbitrary data that can be entered and modified by the end user. These fields are generally considered free-form, i.e. end users can enter whatever kind of value they want to.

DataFields can be initialised with a default value or with a property of the respective type. If a value is used, the persistent value will be initialised with this value. If a property is used, the persistent value will be bound to this property and the initial value will be derived from the property.

```
// Binding initialisation

StringProperty name = new SimpleStringProperty("Hans");
Field.ofStringType(name);

// Value initialisation

Field.ofStringType("Hans");
```

Listing 9 DataField initialisation

The user entry is converted using a value transformer. These helper methods take a String input and convert it to a concrete type, i.e. the generic type of each field. All fields have a default transformer, which can be overridden using the `format()` method.

```
Field.ofDoubleType(5.0)
    .format(d -> Double.parseDouble(d) * 2)
```

Listing 10 Value transformer definition

Value transformations rely on the correct implementation of the transforming methods. The expectation is that the method either returns the transformed value or throws some sort of exception.

Users may define error messages for these value transformations. The messages may be localised.

Property	Purpose	Methods
value	Holds the field's last valid value. Invalid values are not stored in this property.	getValue() valueProperty()
userInput	Holds the current user input. This is an exact representation of the user's input before any transformation or validation happens.	getUserInput() userInputProperty()
validators	Holds a list of all validators.	validate(Validator…)
valueTransformer	Determines the function that is responsible for transforming the user input string into a concrete value.	format(ValueTransformer, String) format(ValueTransformer)

Table 5 DataField properties

6.4.3. StringField

A `StringField` is a concrete implementation of a `DataField`. This type of field handles String input, i.e. any kind of text. Due to the user input being represented as a String, this class has the least amount of restrictions, at least in regard to the value transformation.

By default, this field transforms values using the `String::valueOf` method. This method does not throw any exceptions, thus the transformation will always succeed.

There are two modes for this field. It can be either single-line or multi-line. This is a semantic differentiation which has results on the view layer, but otherwise the fields share a common semantic definition.

Value Type	String
Super Type	DataField
Transformer	String::valueOf

Table 6 StringField information

6.4.4. BooleanField

A `BooleanField` is a concrete implementation of a `DataField`. This type of field handles boolean input, i.e. either truthy or falsy values. In cases where values other than true or false are accepted, it is up to the user to decide which values fall into which category.

By default, this field transforms values using the `Boolean::parseBoolean` method. Like this, only values which are explicitly true are transformed to a true value, all other values just turn into false.

The handling of the `required` case is also different here. Since, by default, any non-null value is considered to fulfil the required condition, `BooleanField` has to be handled differently. Here, only a true value is considered valid, any other value fails this check.

Value Type	Boolean
Super Type	DataField
Transformer	Boolean::parseBoolean

Table 7 BooleanField information

6.4.5. IntegerField

An `IntegerField` is a concrete implementation of a `DataField`. This type of field handles numerical input, more precisely integer values.

By default, this field transforms values using the `Integer::parseInt` method. This method throws an exception if the user input could not be parsed as an integer value, thus failing the type conversion.

Value Type	Integer
Super Type	DataField
Transformer	Integer::parseInt

Table 8 IntegerField information

6.4.6. DoubleField

A `DoubleField` is a concrete implementation of a `DataField`. This type of field handles numerical input, more precisely double values.

By default, this field transforms values using the `Double::parseDouble` method. This method throws an exception if the user input could not be parsed as a double value, thus failing the type conversion.

Value Type	Double
Super Type	DataField
Transformer	Double::parseDouble

Table 9 DoubleField information

6.4.7. SelectionField

A `SelectionField` contains a pre-defined list of available items. End users can then select one or more of those items, depending on the concrete implementation. This type of field cannot be directly modified by the end user, other than changing which elements are selected.

SelectionFields can be initialised with a list of items and an optional initial selection. Alternatively, users can also supply a properties for the items and the selection. This will cause a bidirectional binding between the form and any model classes.

```
// Binding initialisation

ListProperty<String> items = new
SimpleListProperty(FXCollections.observableArrayList("Hello",
"World"));
ListProperty<String> selection = new
SimpleListProperty(FXCollections.observableArrayList("Hello"));
Field.ofMultiSelectionType(items, selection);

// Value initialisation

Field.ofMultiSelectionType(Arrays.asList("Hello", "World"),
Arrays.asList(1));
```

Listing 11 SelectionField initialisation

Internally, the items are defined as a typed list property, thus, a SelectionField can only ever contain values of a single type. For displaying the values, the `toString` method has to be implemented on all elements.

Selection is always based on indices. The user marks the element at a given index as selected and, internally, the element at the given index is added to the list of selected items. This means that while selection is based on indices, getting the selection list returns a list of concrete items.

Property	Purpose	Methods
items	Holds all selectable items. Changes to this also reset the current selection.	items(List, List) items(List, int) items(List) getItems() itemsProperty()
selection	Holds the current user selection. Depending on the selection mode, this is a single object or a list of objects.	getSelection() selectionProperty()
validators	Holds a list of all validators.	validate(Validator…)

Table 10 SelectionField properties

6.4.8. SingleSelectionField

A `SingleSelectionField` is a concrete implementation of a `SelectionField`. As the name implies, it is tasked with handling only single selection.

The field keeps track of a single selected item, rather than a list of selected items.

This also means that selecting a new element removes the selection on the previously selected element. In some cases, an empty selection is desirable. This can be achieved by selecting `null` or the index `-1`.

Users can always change the list of items, along with optionally providing a new selected index.

Validators on this type of fields are expected to be able to handle concrete type of the selectable value. They receive the new selection as an input.

6.4.9. MultiSelectionField

A `MultiSelectionField` is a concrete implementation of a `SelectionField`. This type of field handles multiple selections on a single list of available items. The field thus keeps track of a list of selected items.

Users are provided a `select()` and `deselect()` method which they can use to modify the selection list. The passed indices are validated for correctness and the methods silently fail if the index is invalid.

The list of items can always be changed, along with optionally providing a new list of selected indices.

6.5. Validation

The act of validating user input takes care of ensuring that a form's data is in a valid and useful state. Validation can handle formatting issues as well as semantic issues. Users expect to be able to create rules that cover all their requirements, a flexibility that FormsFX provides. There is support for both pre-defined rules that cover a variety of use cases, as well as custom validation rules.

The factory naming is designed to fit in with the fluent API of the structural elements. This gives users a familiar approach to creating and managing their field validators.

In general, FormsFX validators are typed using generics. This means that each validator can claim support for a given type. Fields can then limit their options of validators to include just the ones that support the field's type. This gives a good amount of flexibility, while still ensuring that no invalid combination is created.

```
Field.ofStringType(".ch").validate(
   StringLengthValidator.exactly(3, "…"),
   RegexValidator.forPattern("^.[a-z]{2}$", "…"),
   CustomValidator.forPredicate(s -> s.contains("xyz"), "…")
)
```

Listing 12 Definition of field validators

Every step of validation results in an instance of a `ValidationResult`. This class contains a boolean that determines success or failure of the validation, as well as any error messages. These messages can optionally be localised. Aggregating a list of error messages is as simple as collecting all message entries of all validation results.

Validation happens in a fixed order. If there is a failure at any step of the way, the following steps are not checked. First of all, the required condition is checked. If a field is marked as required and no entry was given, the validation fails. Then, at least for a `DataField`, the type conversion happens. If the user input could not be converted to the field's concrete type, the validation fails. The next step is going through all the field validators, thus collecting the results and error messages in a list. If all the above validators resulted in a success, the field is marked as valid.

```
List<String> errorMessages = validators.stream()
        .map(v -> v.validate(transformedValue))
        .filter(r -> !r.getResult())
        .map(ValidationResult::getErrorMessage)
        .collect(Collectors.toList());
```

Listing 13 Collection of error messages

6.5.1. Type Validation

Every `DataField` is represented in two ways. First, it is represented as a String, which can be edited by the user. In addition to this, it is also stored as a typed value, i.e. an `IntegerField` has a string-formatted integer and an integer value.

This value transformation also acts as a kind of validation. If the transformation fails, the field is considered invalid as it could not be represented as the concrete value.

Type validators are an extension point for developers. Here, they can pass custom formats, e.g. a transformer that turns strings of numbers with thousands separators into proper `Integers` or `Doubles`.

6.5.2. Required Validation

All fields can be marked as required. For most fields, this simply means that the value must not be empty. `BooleanField` requires a true value, `SingleSelectionField` requires a non-null selection, and `MultiSelectionField` requires a non-empty selection. In the cases where this validation is different from the default it is overridden using the `validateRequired` hook.

6.5.3. Range Validation

There are multiple validators that are only concerned with ensuring that a value lies within a given range. All these validators offer factories for minimum and maximum values, along with methods that only take one limit and imply the other limit. Methods use names like `atLeast`, `upTo`, `shorterThan`, `longerThan`, and `exactly`.

```
private DoubleRangeValidator(double min, double max, String
errorMessage) {
    super(input -> input >= min && input <= max, errorMessage);
}
```

Listing 14 Constructor for a range validator

Numbers (`Integer`, `Double`) can be validated for their value, Strings can be validated for their length, and lists can be validated for their selection size.

6.5.4. RegEx Validation

Strings can be validated using a RegEx validator. The `RegexValdiator` provides a convenient way to check a string against a pattern. The class provides a factory for a user-defined pattern, as well as pre-defined patterns for email addresses, URLs, and alphanumeric values.

6.5.5. Custom Validation

In situations where the provided validators are not enough, developers can also create custom validators. Using the `forPredicate` factory on the `CustomValidator` class, they have the option to pass a `Predicate` that takes the field's transformed value and runs a custom validation on it.

Internally, most validators are subclasses of the custom validators, thus they already use the developer-facing extension point. Ideally, the flexibility created by the pre-defined validators and the custom predicates should make creating new, custom validator classes unnecessary.

6.6. Internationalisation

Business applications are often used in an international context so providing multiple localisations of an applications is a key feature. Handling these locale settings and correctly updating all the necessary components is a difficult and error-prone process, even more so if locale changes need to happen at runtime.

FormsFX is fully *i18n-ready*. It provides mechanisms to translate all displayed values, such as titles, labels, and error messages. Translated values are updated automatically whenever the underlying locale changes using a listener.

Internationalisation is handled using a `TranslationService`. The core task of this service is to take a key and provide a translated value for the current

locale. Each concrete implementation of such a service is responsible for handling locale changes and keeping track of the current locale.

Out of the box, FormsFX offers a `ResourceBundle`-based implementation. It keeps track of the current `ResourceBundle` and notifies listeners whenever it is changed. The translation is based on a lookup of the key in the bundle. This implementation is to be considered a reference implementation. While it covers many use cases, it does not offer proper error handling or fallbacks. A custom `TranslationService` could augment the existing implementation or swap it for something else entirely, e.g. translation using a database.

Internationalisation is handled on the form level. The form automatically passes the localisation down to its children. Thanks to this, adding multi-language capabilities to a form takes only a few lines of code.

```
ResourceBundle rbEN = ResourceBundle.getBundle("demo.demo-
locale", new Locale("en", "UK"));
ResourceBundleService rbs = new ResourceBundleService(rbEN)

Form.of(…).i18n(rbs);
```

Listing 15 Adding internationalisation to a form

Multi-language is not mandatory. Applications can still be developed using a single locale. Developers can then use the exact same API so there is no divergence in the two approaches. The switch, however, is absolute. Forms with mixed content, i.e. some localised and some non-localised strings, are not currently possible.

6.7. Testing

There is a test suite covering the most important aspects of the model layer. These tests ensure correct functionality of the core components, like structural elements and validators. The tests achieve a coverage of roughly 80%, i.e. they cover all the core functionality and most of the extended functionality.

7. View

7.1. Renderers

Renderers are the core components responsible for turning the modelled data into a visible user interface. They take the structural elements and use their information to present end users with a well-designed and user-friendly form.

Each structural level has a different renderer, with the field level offering an extension point for developers where they can swap out the default renderer for a custom control of their own.

7.1.1. Form

The `FormRenderer` class is, in most cases, the only class that developers directly interact with. It renders form specifics and redirects rendering for groups and sections to their respective renderers (see *7.1.2.*)

Internally, a form is displayed as a `VBox` where groups and sections are added as child nodes in a vertical fashion.

```
Form formModel = Form.of(…);
getChildren().add(new FormRenderer(formModel));
```

Listing 16 Form inclusion in a UI

7.1.2. Group and Section

Just as on the semantic level, groups and sections are also handled differently on the view level. Groups represent a looser coupling of fields by only visually grouping the elements inside a rectangle.

Sections, on the other hand, use a `TitledPane` to offer a more visually obvious binding. Due to this, sections can also carry a title and be collapsed by the end user.

Internally, both `GroupRenderer` and `SectionRenderer` inherit from `SectionRendererBase`, which defines shared characteristics between the two subtypes, such as grid columns and the way fields are added to the group. For sections, the grid is then added to a `TitledPane`, which handles collapsing in the UI.

The concrete rendering for a field is not handled by the group, but instead redirected to the field's renderer (see *7.1.3.*)

7.1.3. Field

Fields do not have a general renderer. Instead, each field has a specific default renderer that is able to handle the field's data. Developers can then decide to override this default renderer with another compatible implementation, possibly even a custom implementation of their own.

```
Field.ofMultiSelectionType(…, …)
  .render(new SimpleCheckBoxControl<>())
```

Listing 17 Changing of the default renderer on a field

This works by only initialising the control during rendering. In the end, this offers a great deal of flexibility combined with type safety.

7.2. Grid Layout

A consistent layout is key to a visually pleasing form. Due to the different types of data that need to be handled within a form and their respective controls, achieving such a layout is often not an easy task.

In FormsFX, this is achieved by using a `GridPane` to layout children of the controls. Every field has a definition for `colspan`, i.e. a value that determines the number of available columns and their width. The pre-defined controls then all give two columns to the label, while the remaining columns are reserved for the editable controls.

```
int columns = field.getSpan();

for (int i = 0; i < columns; i++) {
    ColumnConstraints colConst = new ColumnConstraints();
    colConst.setPercentWidth(100.0 / columns);
    getColumnConstraints().add(colConst);
}
```

Listing 18 Column definitions for a simple control

Controls are then added to the `Group` grid, which is made up of 12 columns. This grid handles different situations, namely one where the `colspan` values always result in 12 columns per row, but also one where this is not the case. In this situation, the grid just leaves empty space at the end of a row and moves the control to the next row.

```
for (Field f : element.getFields()) {
    int span = f.getSpan();

    if (currentColumnCount + span > COLUMN_COUNT) {
        currentRow += 1;
        currentColumnCount = 0;
    }

    SimpleControl c = f.getRenderer();
    c.setField(f);

    grid.add(c, currentColumnCount, currentRow, span, 1);

    currentColumnCount += span;
}
```

Listing 19 Column handling in a group

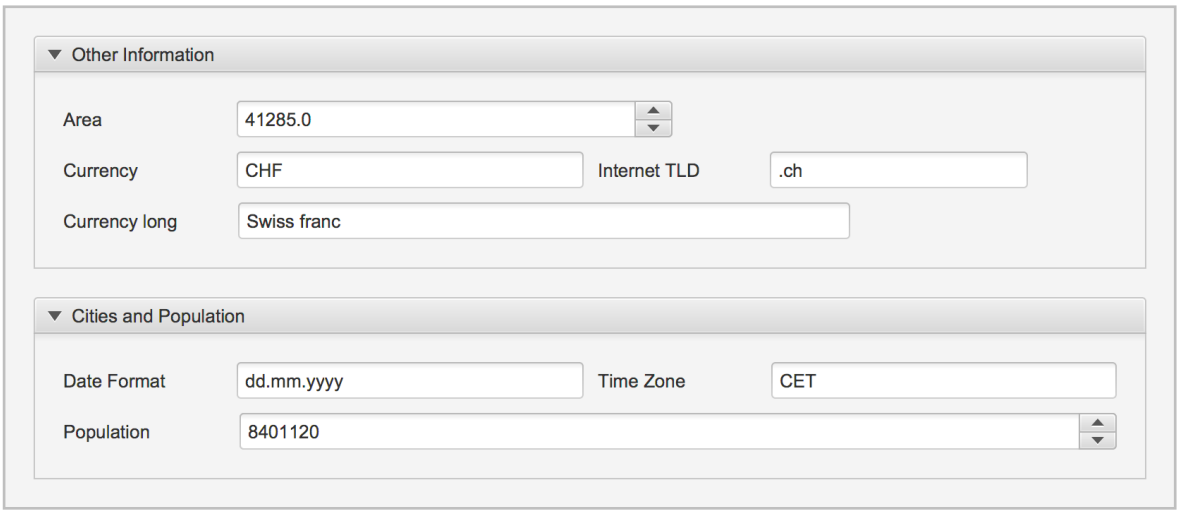

Figure 5 Grid layout with different column spans

Thanks to the `Group` and `Field` grids, a consistent layout can be achieved throughout the whole form, regardless of how developers decide to structure their forms.

In order to make the grid layout more transparent to developers, a custom `ColSpan` enum is provided with proportions instead of numbers, e.g. `ColSpan.HALF` represents 6 columns. The 12 column grid offers many possible divisions, such as halves, thirds, quarters, and sixths.

7.3. Custom Controls

On the view level, each field can be represented by a custom control. This is an ideal extension point for developers where they can easily integrate their own controls. In order to provide a proper out-of-the-box experience, FormsFX provides a collection of reference implementations. These simple controls handle the basic tasks of form controls, combined with the features and possibilities of the backing fields.

7.3.1. SimpleControl

All controls share the `SimpleControl` base class. This abstract class holds information which is to be available in every control. Most importantly, it has a reference to a field, the type of which is defined using generics. Each control can state which level of specificity of fields it decides to support.

The base class does not contain any UI nodes, i.e. it could not be rendered by itself. It is up to the concrete classes to determine, how the UI structure needs to be set up for the control to work ideally.

The custom controls use pseudo classes to handle state-specific styling. These classes react to state changes on the fields, e.g. when its validity changes from valid to invalid, or vice versa. These classes help with structuring the CSS code as they create a clear division between element and state. Once a pseudo class has been set, CSS rules can target elements in a specific state, e.g. `.control:invalid`.

```
REQUIRED_CLASS = PseudoClass.getPseudoClass("required");
INVALID_CLASS = PseudoClass.getPseudoClass("invalid");
CHANGED_CLASS = PseudoClass.getPseudoClass("changed");
DISABLED_CLASS = PseudoClass.getPseudoClass("disabled");
```

Listing 20 Pseudo classes used for custom controls

Using `pseudoClassStateChanged` — a method defined on the `Node` class — pseudo classes can easily be added to or removed from an object.

Enabling or disabling the four pseudo classes depends on certain properties of the `Field` classes. In general, the state is set when the control is initialised, while listeners handle changes to the properties.

```
setupValueChangedListeners() {
 field.validProperty().addListener(() ->
   updateStyle(INVALID_CLASS, !newValue));
 field.requiredProperty().addListener(() ->
   updateStyle(REQUIRED_CLASS, newValue));
 field.changedProperty().addListener(() ->
   updateStyle(CHANGED_CLASS, newValue));
 field.editableProperty().addListener(() ->
   updateStyle(DISABLED_CLASS, !newValue));
}
```

Listing 21 Listeners to set and update pseudo classes

Every control has a tooltip to show more information about the field. This includes pre-defined messages that aid the end user's understanding of the field, but also any error messages that arise from the user input. Depending on the controls model of interaction, tooltips are either shown when the editable control is focused, or when the editable control is hovered. These two options cover most general use cases and thus offer much flexibility.

In general, concrete control implementations only determine when the tooltip should be displayed. The logic behind creating the tooltip and positioning it below a given control lies in the `SimpleControl` class so a consistent user experience can be achieved.

As with most JavaFX controls, the simple controls offer an easy way to add custom styling by taking an ID and a list of style classes, which can then be used as a styling hook. This adds an additional layer of flexibility for the developer.

7.3.2. SimpleTextControl

This implementation of a `SimpleControl` handles text input, i.e. a `StringField`. The control creates a `Label` and `TextField` and renders them in a grid.

The speciality of this control is, that it is not only used for `TextField`, but for `TextArea` as well. This distinction is based on the multiline property of the field and can even be changed dynamically at runtime.

In order to achieve this a managed property is bound with the visibility property for the `TextField` and `TextArea`. The managed property removes the node completely from the view and does not just hide it. This way all the layout settings also get updated.

```
editableArea.managedProperty().bind(editableArea.visibleProperty());
editableField.managedProperty().bind(editableField.visibleProperty());
```

Listing 22 Managed property binding to switch between nodes

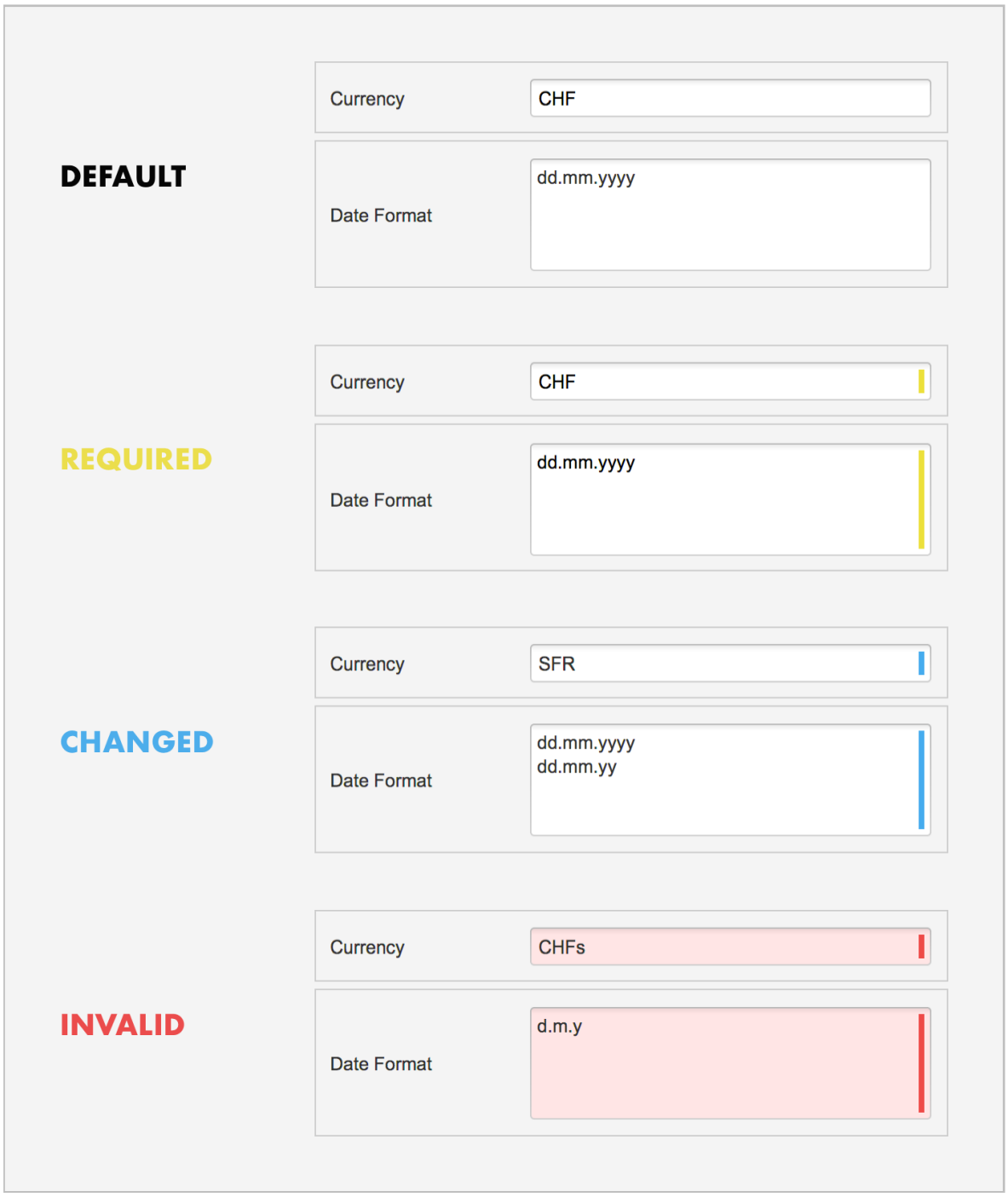

Figure 6 Default styling for SimpleTextControl in different states

The fact that this control supports two editable controls based on a condition, along with the default read only mechanism, leads to a more specific binding. This binds the text field's and text area's visibility properties to a binding, created using both conditions, i.e. multiline and editable.

```
editableArea.visibleProperty().bind(Bindings.and(field.editable
Property(), field.multilineProperty()));
editableField.visibleProperty().bind(Bindings.and(field.editabl
eProperty(), field.multilineProperty().not()));
```

Listing 23 Visibility property binding to hide and show nodes

For the read only mechanism, `TextField`, `TextArea`, and a `Label` are added to a `StackPane`. Depending on the visibility property the field or area is hidden and the label will be shown, which will always show the current user input in a non-editable label.

7.3.3. SimpleNumberControl

For the number types — `Integer` and `Double` — there need to be two separate custom controls so that each type can be rendered individually. However, since most of the code is identical, those parts are shared in a base abstract class `SimpleNumberControl`.

This control creates a `Label` and `Spinner` and renders them in a grid. The read only mechanism works by hiding or showing either the `Spinner` or the `Label`, which are both included in a `StackPane`. In order to achieve this, the visibility properties of the elements are bound to the field's editable property.

In order to improve the user experience, both numeric controls can be incremented or decremented with the up and down keys, respectively.

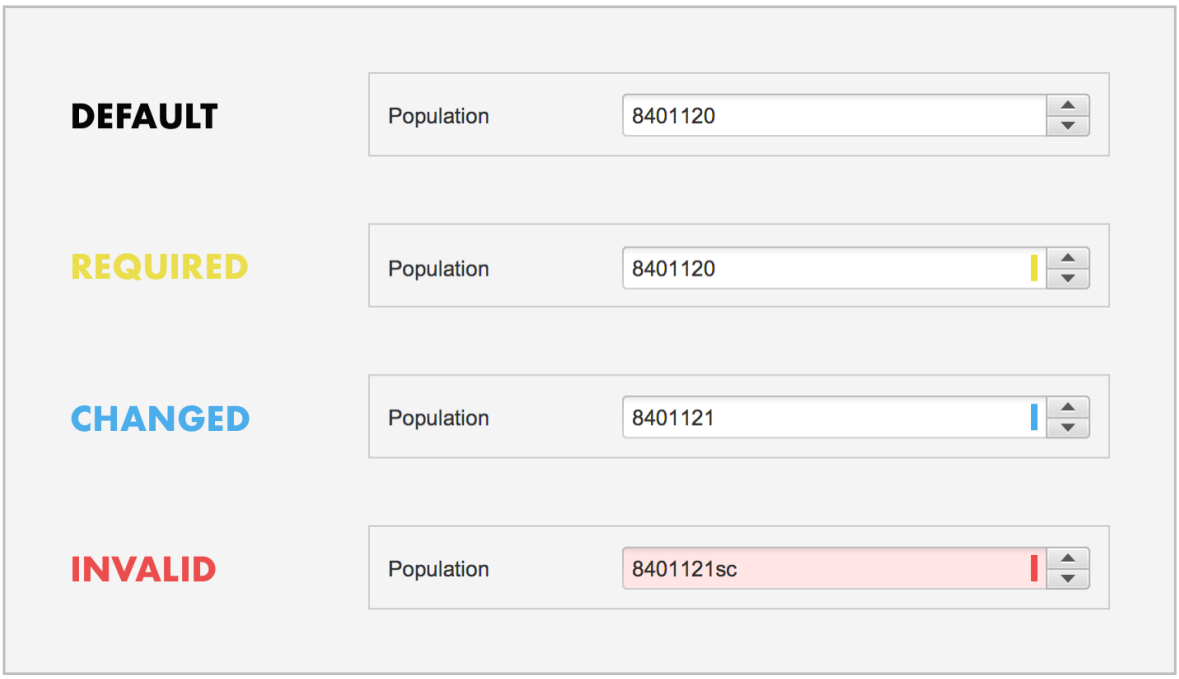

Figure 7 Default styling for SimpleNumberControl in different states

7.3.4. SimpleIntegerControl

This implementation of a `SimpleControl` handles numerical data, specifically integral numbers, i.e. an `IntegerField`. The control's spinner is initialised using integer values. A concrete minimum or maximum is not defined, developers are instead encouraged to use range validators (see *6.5.3.*)

```
editableSpinner.setValueFactory(new
SpinnerValueFactory.IntegerSpinnerValueFactory(Integer.MIN_VALU
E, Integer.MAX_VALUE, field.getValue()));
```

Listing 24 Creating an Integer-type Spinner

7.3.5. SimpleDoubleControl

This implementation of a `SimpleControl` handles numerical data, specifically double precision numbers, i.e. a `DoubleField`. The control's spinner is initialised using double values. A concrete minimum or maximum is not defined, developers are instead encouraged to use range validators (see *6.5.3.*)

7.3.6. SimpleBooleanControl

This implementation of a `SimpleControl` handles boolean data, i.e. a `BooleanField`. This control creates a `Label` and a `CheckBox`. The `CheckBox` is added to a `VBox` so that state stylings can be applied on the `VBox` instead of the `CheckBox` or the whole control in order to keep the the overall style of the form uniform. The read only mechanism works by enabling or disabling the `CheckBox`.

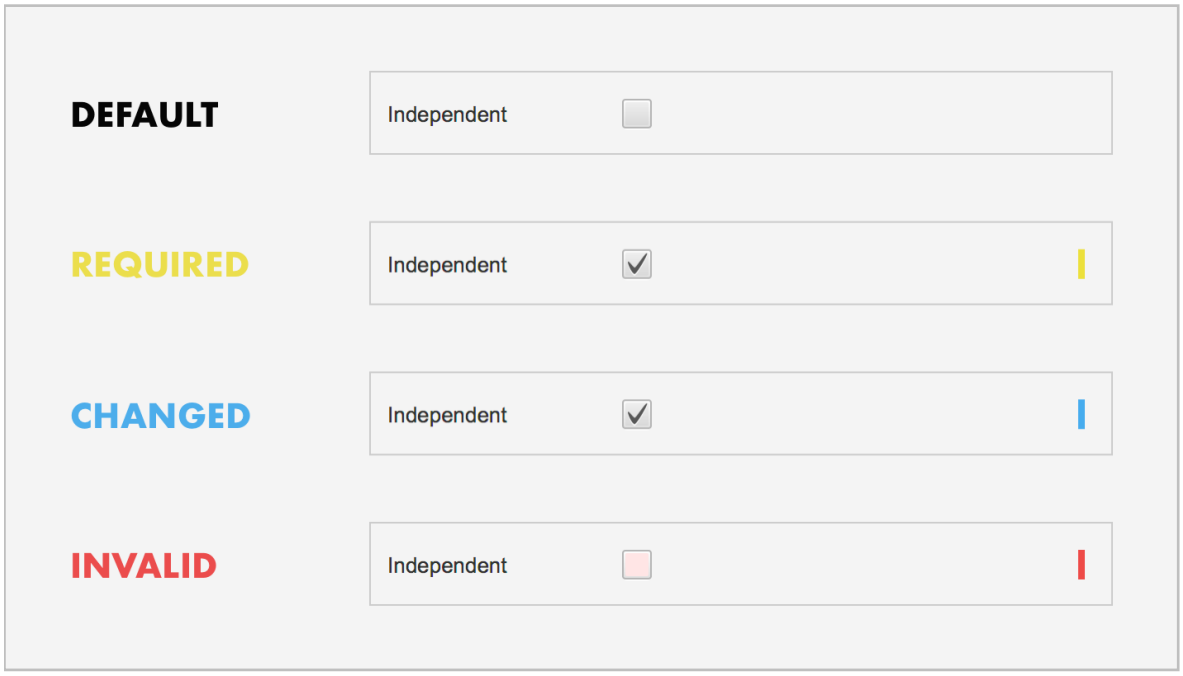

Figure 8 Default styling for SimpleBooleanControl in different states

7.3.7. SimpleListViewControl

This implementation of a `SimpleControl` handles lists in multi selection mode, i.e. a `MultiSelectionField`. The control renders a `Label` and `ListView`.

Since this control relies on string representation of the lists contents, the binding from the view to the model and vice versa is handled with listeners instead of bindings like with most other controls.

Another difference to the other controls is that since this control does not have an editable control that can be focused, at least not like other controls, the `Tooltip` has to be shown when the user hovers over the complete `List-`

`View` and not only on one item in the list. Because of that the `Tooltip` gets toggled on mouse enter and leave.

```
listView.setOnMouseEntered(event -> toggleTooltip(listView));
listView.setOnMouseExited(event -> toggleTooltip(listView));
```

Listing 25 ListView Tooltip handling

The read only mechanism for this control works by enabling or disabling the `ListView`.

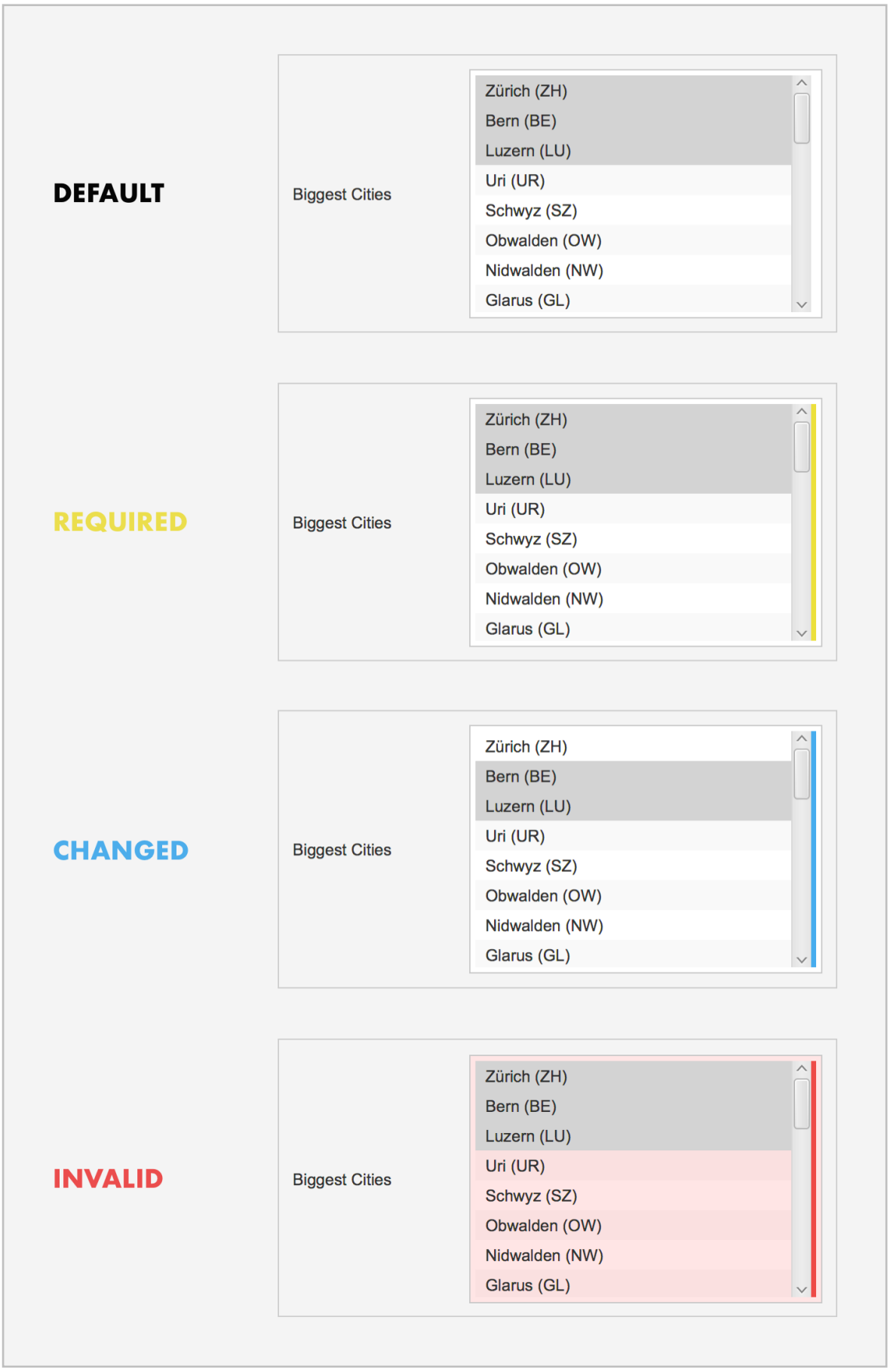

Figure 9 Default styling for SimpleListViewControl in different states

7.3.8. SimpleComboBoxControl

This implementation of a SimpleControl handles lists in single selection mode, i.e. a `SingleSelectionField`. The control renders a `Label` and `ComboBox`, which are added to a `StackPane` so that they can be hidden and shown to achieve the read only mechanism.

Since this control does not have an editable control the `Tooltip` will be only shown, when the user hovers over the `ComboBox`. This is solved similar to the `Tooltip` handling of `SimpleListViewControl` (see *7.3.7.*)

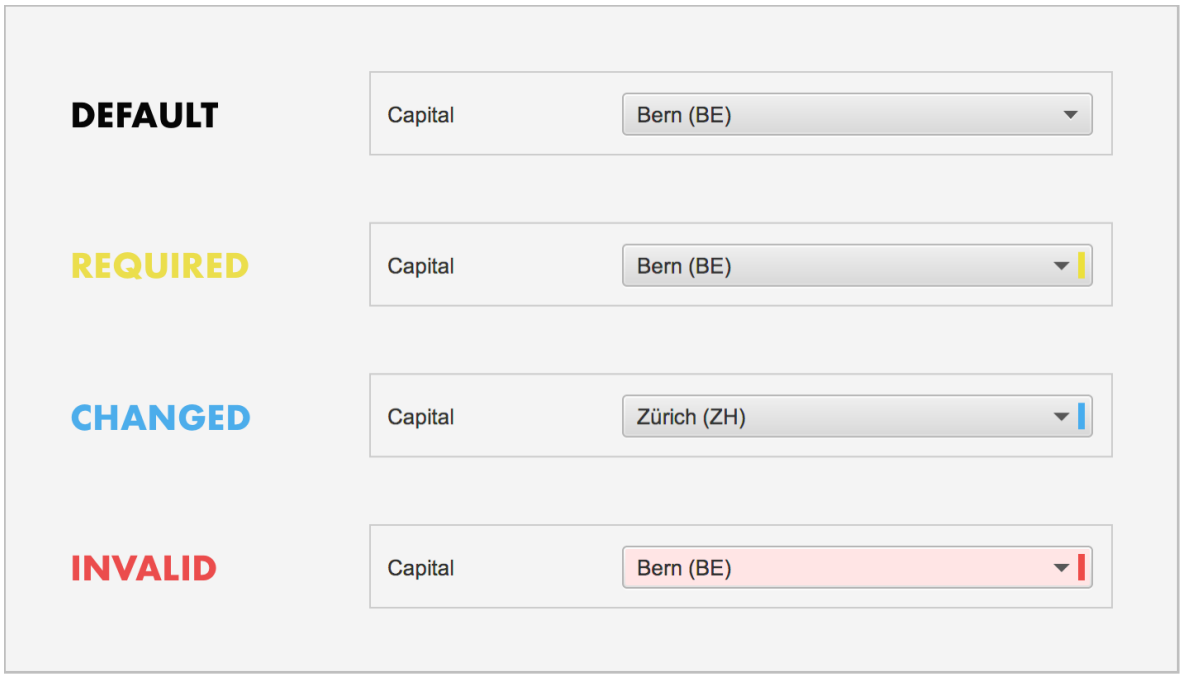

Figure 10 Default styling for SimpleComboBoxControl in different states

7.3.9. SimpleCheckBoxControl

This implementation of a SimpleControl handles lists in multi selection mode, i.e. a `MultiSelectionField`. The control renders renders a `Label` and a list of `CheckBoxes`, which are added to a `VBox` for state styling purposes. The read only mechanism here works enabling or disabling the `CheckBoxes` whenever the editable property of the field changes.

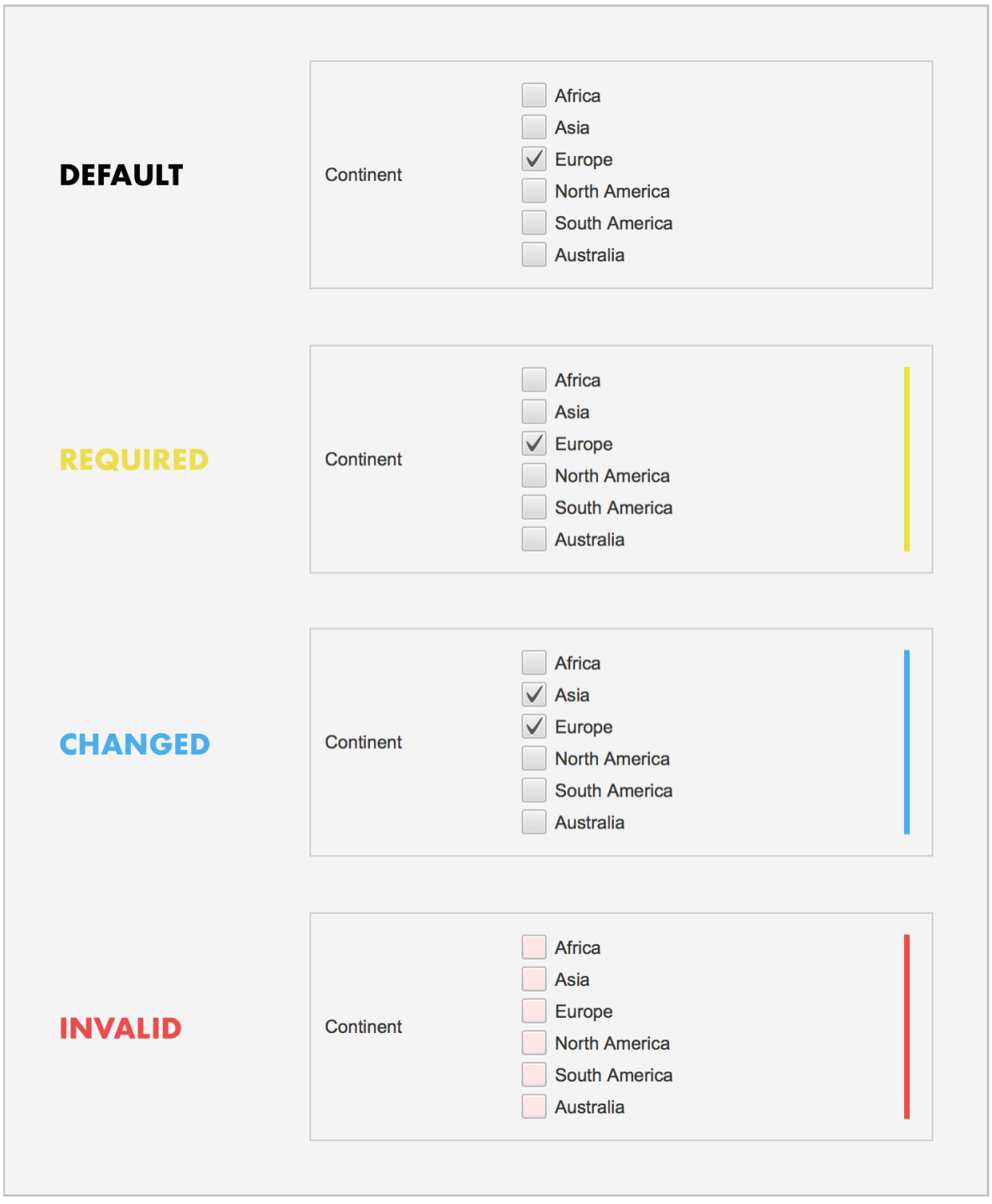

Figure 11 Default styling for SimpleCheckBoxControl in different states

With this control there is also a difference for the handling when the items of the field changes. It is not a simple task of setting up a binding, but instead means that all checkboxes have to be recreated. Since the `CheckBoxes` have to be redrawn, they also need the new bindings as well as the new event handlers.

```
field.itemsProperty().addListener((observable, oldValue,
newValue) -> {
    createCheckboxes();
    setupCheckboxBindings();
    setupCheckboxEventHandlers();
});
```

Listing 26 Handling of items change of field

7.3.10. SimpleRadioButtonControl

This implementation of a SimpleControl handles lists in single selection mode, i.e. a `SingleSelectionField`. The control creates a `Label` and a list of `RadioButtons`, which are added to a `VBox` for state styling purposes. The read only mechanism works by enabling or disabling the `RadioButtons` when the editable property of the field changes.

When the items on the field change, the `RadioButtons` have to be redrawn. This means new `RadioButtons` are created, therefore they need to be bound again and the event handlers have to be set up again (see *Listing 24*.)

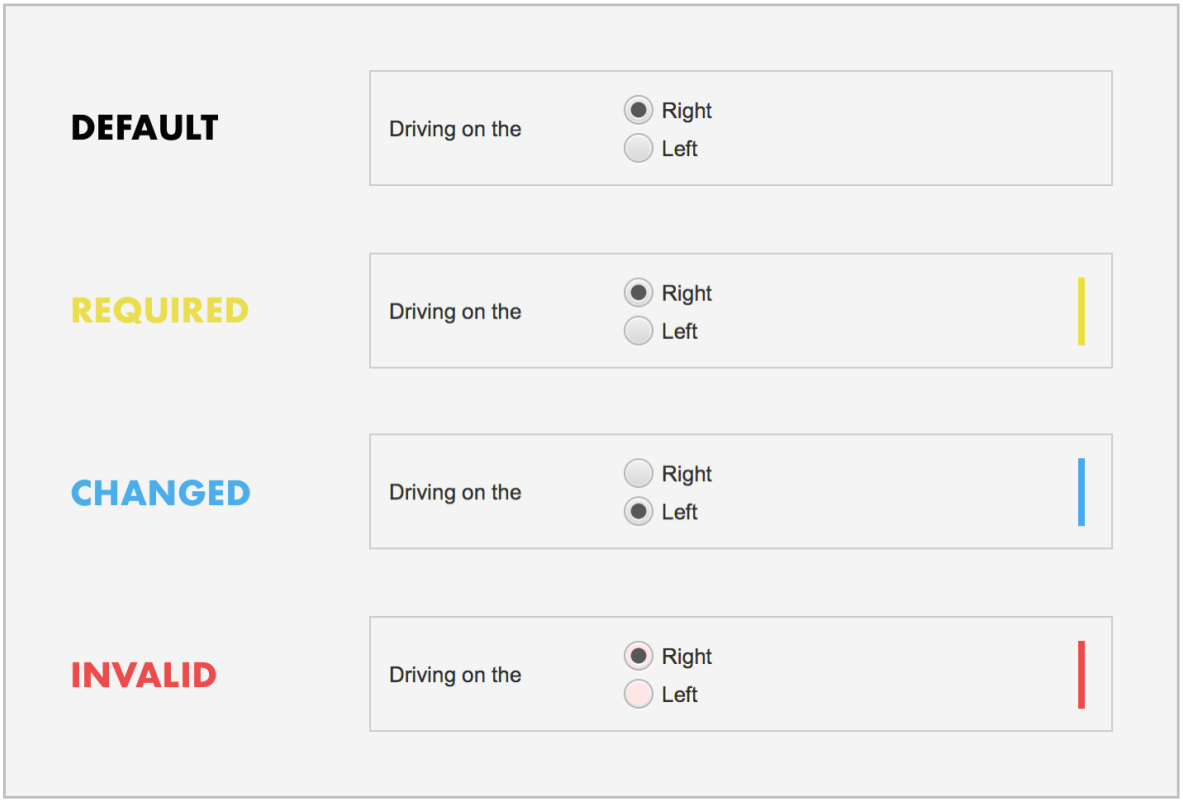

Figure 12 Default styling for SimpleRadioButtonControl in different states

7.4. CSS Styling

FormsFX uses very little custom styling, mainly to achieve consistency between some of the components or to handle the visualisation of state changes.

In some cases, such as with `TextField` and `TextArea`, JavaFX's default styling is different to the point that the controls feel unrelated, despite serving a similar purpose, which is why `TextArea` had to be adapted to look more like the default `TextField`.

In other cases, however, it is a matter of adding styling to otherwise plain components, such as with `Group` and `Section`. Since sections are wrapped in a `TitledPane`, they reside in a clear container, which is not true for groups. Due to this, groups were given some styling to make their containment more clear.

Most of the other styling handles state changes so most of the selectors rely on custom pseudo classes (see *7.3.1.*) The states are represented using a coloured bar on the far right side of the control, where yellow represents required, blue represents changed, and red represents invalid. Additionally, invalid controls are highlighted with a light red background.

The custom styling is combined in `view/style.css`. Developers can simply add this stylesheet to their own applications.

7.5. Testing

There is a test suite covering the most important aspects of the view layer. These tests ensure correct functionality of the core components, like renderers and controls. The tests achieve a coverage of roughly 90%, i.e. they cover all the core functionality and most of the extended functionality.

8. Conclusion

At the end of this bachelor's project, FormsFX solves many of the goals it was set out to solve. Already, developing and managing a form is a much simpler task, both compared to standard JavaFX, as well as the competition. Developers can now focus their work much clearer on the tasks that are actually important, like defining the structure and the semantics of the form, rather than wasting time on tedious and repetitive work.

Much of the complexity in this project came from the separation between model and view. Not always was it clear, which layer would be responsible for a given task so this lead to some discussion. Even then, many discussions were held about semantic issues or naming conventions. All of this in an attempt to create a well-structured API that developers can quickly learn and start using.

The end result is a very flexible and extensible solution. Especially the synergy between model and view and the way developers can extend the provided functionality will lead to much more efficiency for the developer. Whether they want to use an out-of-the-box experience and use the provided controls or create an entirely new experience with custom controls, custom validation rules, and more powerful internationalisation services, all the required functionality and extensibility is provided.

9. Evolution Scenarios

9.1. Tab Indices

Controlling the user's flow through a form is an important part in creating a user-friendly form. The key part in this is specifically designing the tab navigation, i.e. determining which fields follow which.

In Java, this is handled using things like `FocusTraversalPolicy` or `TraversalEngine`, both of which are not currently recommended to use in JavaFX 8, and will likely arrive with JavaFX 10. At the point of writing this report, support for custom tab order was not available, which is why it was left out of the project.

The current implementation determines the tab order based on the order in which fields have been added to the form.

9.2. Tables

There was much discussion on the topic of tables throughout this project. Tables can be a vital part of forms-based applications so their inclusion in FormsFX and possible approaches to the problem were considered. It was eventually decided to postpone work on this issue and focus on other areas.

9.3. Business Controls

In parallel to the FormsFX project, another project called «MultiDeviceBusinessControls» was conducted as a Bachelor's project. During the project, the option of combining the two projects to some degree was kept open. A first push towards an integration was made by the Business Controls team, but the idea was eventually pushed back and not considered in-scope for the FormsFX project anymore.

It is, however, very much possible to integrate the other project, as well as other business controls into the FormsFX project. In the long run, it could

even be desirable to maintain a repository of controls that are compatible with the FormsFX API. These controls could be new developments, or simply wrappers for existing controls.

10. References

10.1. List of Tables

10.2. List of Figures

10.3. List of Listings

11. Honesty Declaration

It is hereby declared that the contents of this report, unless otherwise stated, have been authored by Sacha Schmid and Rinesch Murugathas. All external sources have been named and quoted material has been attributed appropriately.

Windisch, 18 August 2017

Date and Location

Sacha Schmid

Rinesch Murugathas