

Table of contents

Getting started with JavaFX.....	2
Study this first.....	2
Real world examples.....	2
child controllers.....	2
sometimes need css class of inner component.....	4
a custom component containing another custom component.....	4
the accompanying java for the custom component.....	5
Use of a custom component.....	5
expression binding.....	6
binding.....	6
SimpleObjectProperty.....	7
setup observing to keep values and GUI in sync.....	8
custom cellFactory.....	8
a custom CellValueFactory.....	9
a ColorPicker as content of a cell.....	10
A value is incorrect, the value field is styled accordingly. As soon as the typed value becomes correct the “error” style is removed.....	10
FXCollections.....	11
swing integration.....	11
built-in dialogs.....	12
FileChooser.....	12
packaging: javafx-maven-plugin.....	13
use exec-maven-plugin for running / debugging.....	13

Getting started with JavaFX

This document points you to excellent reading material by oracle on JavaFX. Furthermore you will find code examples from my own experience with JavaFX in practice.

Study this first

<https://docs.oracle.com/javafx/2/> The tracks here are very good, below I mention the ones you should at least read.

https://docs.oracle.com/javafx/2/layout/size_align.htm

https://docs.oracle.com/javafx/2/api/javafx/fxml/doc-files/introduction_to_fxml.html

<https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html>

<https://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm>

<https://docs.oracle.com/javase/8/javafx/api/javafx/collections/FXCollections.html>

<https://docs.oracle.com/javase/8/javafx/api/javafx/beans/binding/Bindings.html>

Also worth mentioning are <http://jfxtras.org/> and <http://fxexperience.com/controlsfx/>

Real world examples

Below examples are taken from <https://github.com/eduardddrenth/iText-GUI/tree/workshopsolution>

Running:

```
mvn verify jfx:jar exec:exec
```

then you may want to import the file `src/test/resources/config/styling.properties`.

child controllers

Naming of classes and fxml files is tricky here, **stick to the (naming)conventions** or expect exceptions that are not easy to understand.

Root element of the fxml file to include, note the controller class

```
<AnchorPane xmlns:fx="http://javafx.com/fxml"
fx:controller="com.vectorprint.vectorprintreportgui.TableViewController">
```

The include element in the parent fxml file, note the `fx:id`, the name of the field in the parent controller annotated with `@FXML` must start with this id and end with "Controller".

```
<fx:include source="TableView.fxml" fx:id="tableView" />
```

The reference to the child controller in the parent controller

```
@FXML
private TableViewController tableViewController;
```

The image on the next page shows a part that originates from an `fx:include`

Build VectorPrint report configurations

create

stylers/conditions

stylesheet

pdf

xsd data mapping

settings xsd

syntax

help

errors

validate css

File

general options

table of contents

☒ footer

☐ pdf 1/a

☐ debug

☒ pre and post style

reset

condition to determine when to style or not

styleClass / key

bigbold

X

Font{parame...

?

show configuration

add

(re)build stylesheet

special styleclasses

prestylers

poststylers

pagestylers

Configuration

Font

key	type	value	default	reset	which class declares (uses) the parameter
color	java.awt.Color	Black		reset	com.vectorprint.report.itext.style.stylers.Font
conditions	java.lang.String				com.vectorprint.report.itext.style.stylers.AbstractStyler
encoding	com.vectorprint.r...	WINANSI		reset	com.vectorprint.report.itext.style.stylers.Font
family	java.lang.String	Helvetica		reset	com.vectorprint.report.itext.style.stylers.Font
size	java.lang.Float	12.0		reset	com.vectorprint.report.itext.style.stylers.Font
style	com.vectorprint.r...	normal		reset	com.vectorprint.report.itext.style.stylers.Font
styleafterad...	java.lang.Boolean			reset	com.vectorprint.report.itext.style.stylers.AbstractStyler
size	java.lang.Float	12.0		reset	com.vectorprint.report.itext.style.stylers.Font
color	java.awt.Color	Black		reset	com.vectorprint.report.itext.style.stylers.Font
styleafterad...	java.lang.Boolean			reset	com.vectorprint.report.itext.style.stylers.AbstractStyler
style	com.vectorprint.r...	normal		reset	com.vectorprint.report.itext.style.stylers.Font
encoding	com.vectorprint.r...	WINANSI		reset	com.vectorprint.report.itext.style.stylers.Font
family	java.lang.String	Helvetica		reset	com.vectorprint.report.itext.style.stylers.Font
conditions	java.lang.String				com.vectorprint.report.itext.style.stylers.AbstractStyler

sometimes need css class of inner component

You will need the JavaFX css reference when you develop stylesheets. There you will for example find that a **TextArea** has an inner “content” that you need when you want to change the background color.

```
.searching .content {
  -fx-background-color: #eeeeee;
}
```

a custom component containing another custom component

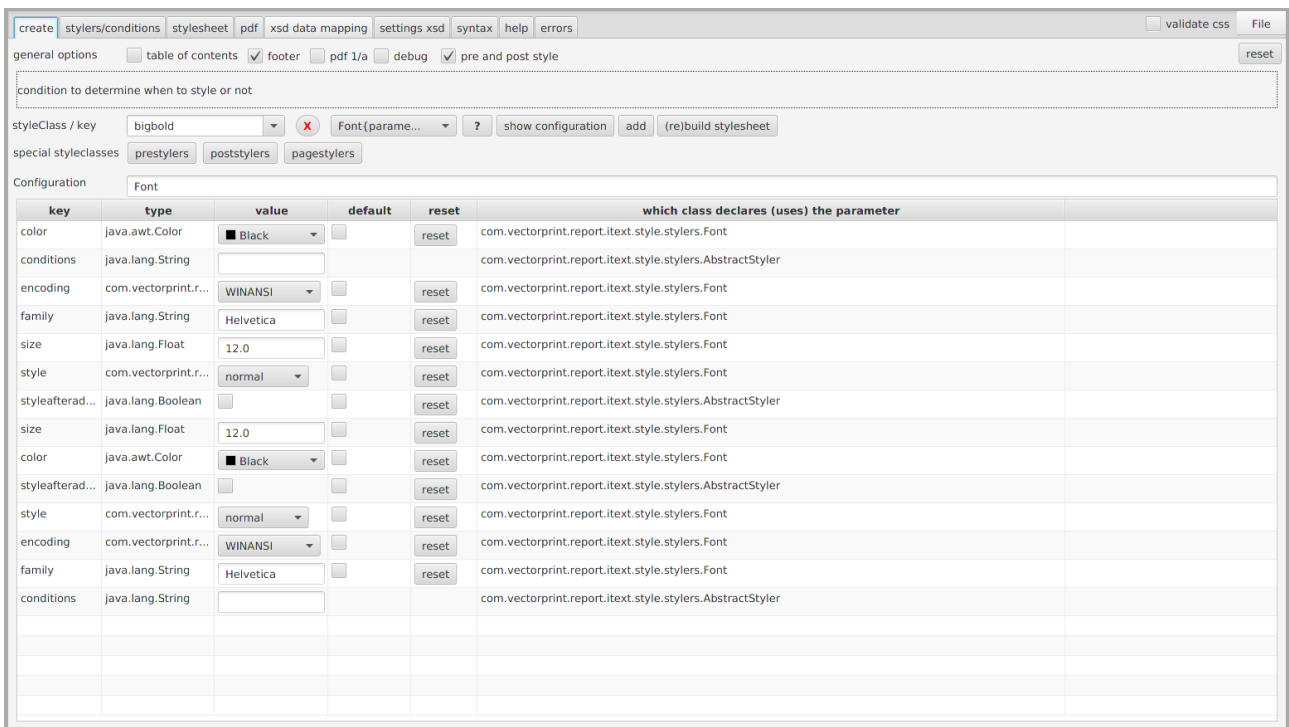
More flexible compared to `fx:include`, you can create custom components consisting of `fxml` and accompanying `java`. You can extend an existing JavaFX Node, add content, add `css` classes, publish properties, etc. Also custom components can be imported in SceneBuilder, to use in designing. **Note** that this importing makes a hardcopy of the jar containing your custom components. **Note the Anchor properties** that pin the bounds of a Node to the bounds of the wrapping `AnchorPane`. This is a great feature when resizing windows.

```
<fx:root type="javafx.scene.layout.AnchorPane"
xmlns:fx="http://javafx.com/fxml">

    <TabPane tabClosingPolicy="UNAVAILABLE" styleClass="xmlareatabs"
AnchorPane.topAnchor="12" AnchorPane.bottomAnchor="0" AnchorPane.leftAnchor="0"
AnchorPane.rightAnchor="0" >

        <Tab text="xml text (searchable)" >
            <SearchableTextArea fx:id="searchText" AnchorPane.bottomAnchor="0"
AnchorPane.leftAnchor="0" AnchorPane.rightAnchor="0" AnchorPane.topAnchor="0" />
        </Tab>
    </TabPane>
</fx:root>
```

This image show a resized window with everything nicely sized and placed due to Anchors.



the accompanying java for the custom component

The java for a custom component is pretty straightforward, extend a Node, some obligatory code in the constructor for the wiring part. Furthermore getters and setters you define can directly be used in fxml and will be recognized by SceneBuilder and NetBeans.

```
public class XmlArea extends AnchorPane {

    @FXML
    private SearchableTextArea searchText;

    @FXML
    private TextFlow xmlhighlight;

    public XmlArea() {
        FXMLLoader fxmLoader = new
FXMLLoader(getClass().getResource(XmlArea.class.getSimpleName() + ".fxml"));
        fxmLoader.setRoot(this);
        fxmLoader.setController(this);

        try {
            fxmLoader.load();
        } catch (IOException exception) {
            throw new RuntimeException(exception);
        }
    }
}
```

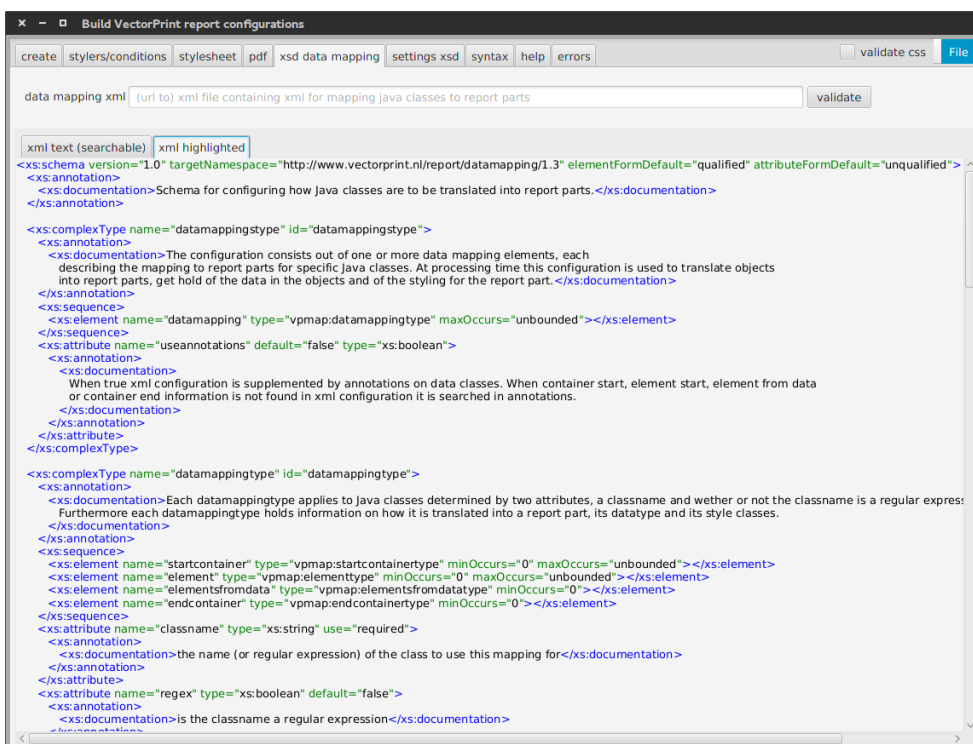
Use of a custom component

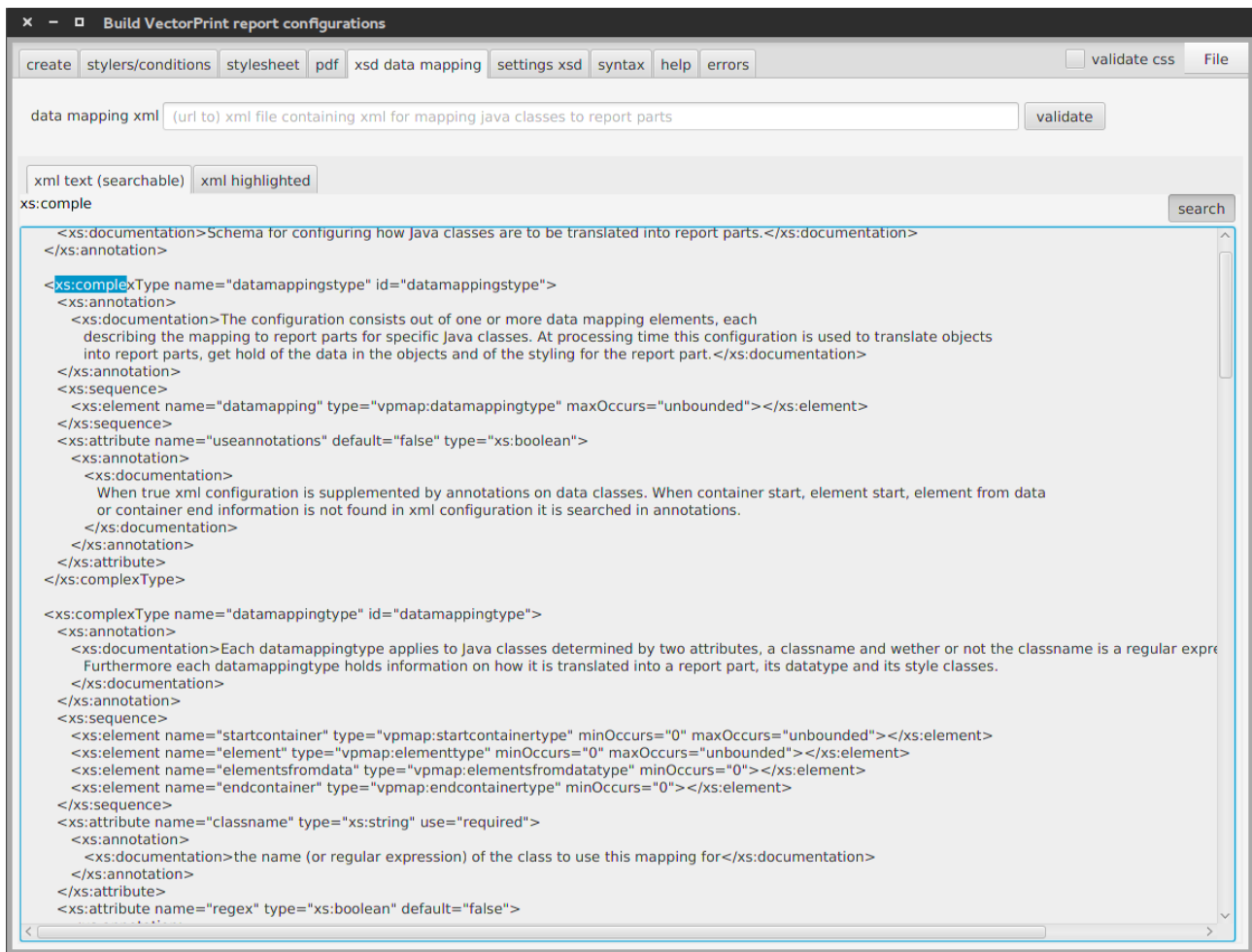
Self explanatory uses of custom components. **Note** the use of custom property “editable”.

```
<SearchableTextArea fx:id="stylesheet" AnchorPane.bottomAnchor="0"
AnchorPane.leftAnchor="0" AnchorPane.rightAnchor="0"
AnchorPane.topAnchor="56" />
```

```
<XmlArea editable="false" fx:id="datamappingxsd" AnchorPane.bottomAnchor="5"
AnchorPane.leftAnchor="5" AnchorPane.rightAnchor="5" AnchorPane.topAnchor="60"/>
```

Below you see images of the XmlArea component. It contains two tabs, one with highlighted xml, the other with the same xml, searchable and optionally editable. The searchable xml is a custom component itself.





Search functionality of the custom component in action, grayed out, readonly text, highlighted match, showing green search text.

expression binding

You can use expressions in fxml to determine values of properties. **Note** this is a binding! When the width of the parameterizableTable changes the column width changes along. **Note** do not add a space before or after the “\${“ or “}”, you will get a weird exception.

```
<TableColumn fx:id="sUpDown" text="upDown" sortable="false"
prefWidth="${parameterizableTable.width * 0.2}" />
```

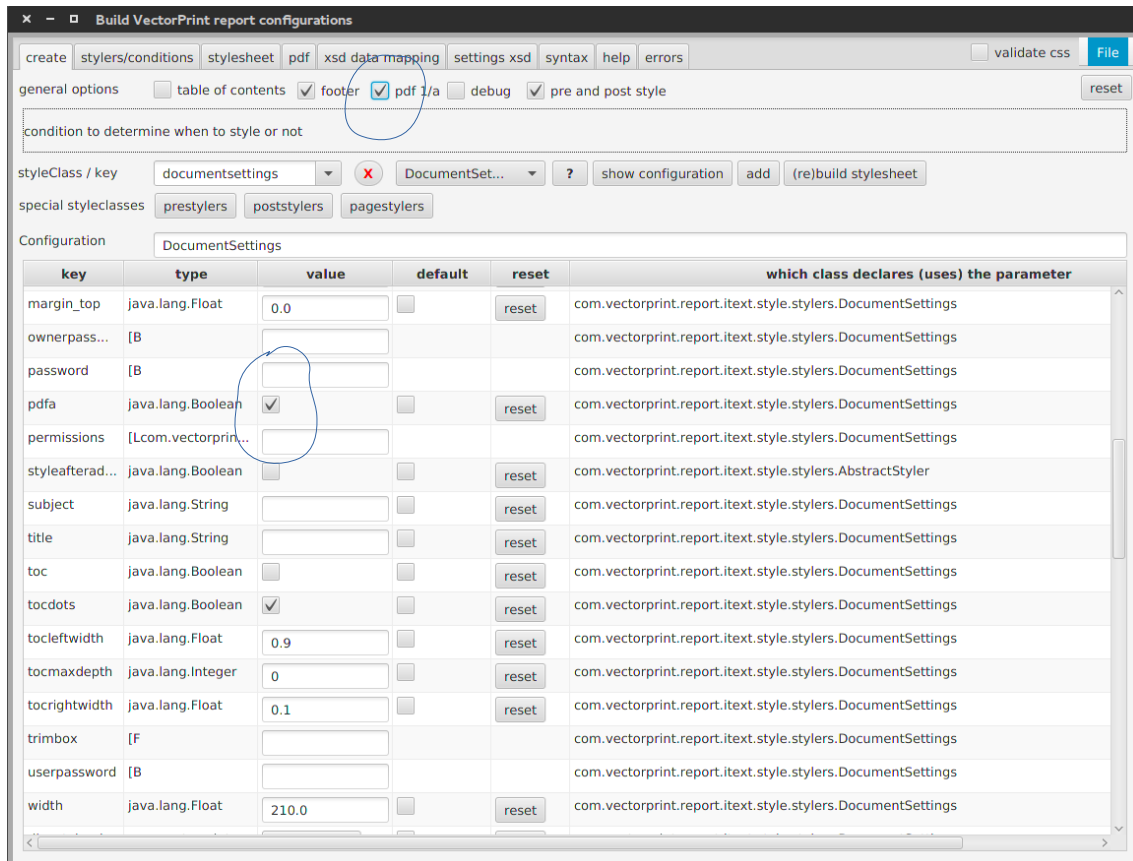
binding

One of the neat possibilities of binding is that you can use SimpleObjectProperty to wrap your own objects and make them bindable. All you need to do is call #set() at the leading side (you cannot call #set() on a bound property) and the bound properties will be updated. **Note** that the binding features in JavaFX is just Java, you can use it in any Java solution, the same holds for other features.

```
private final ObjectProperty<Parameterizable> currentParameterizable = new
SimpleObjectProperty<>();
```

```
tableViewController.getCurrentParameterizable().bind(currentParameterizabl
e);
```

An example of bidirectional binding



SimpleObjectProperty

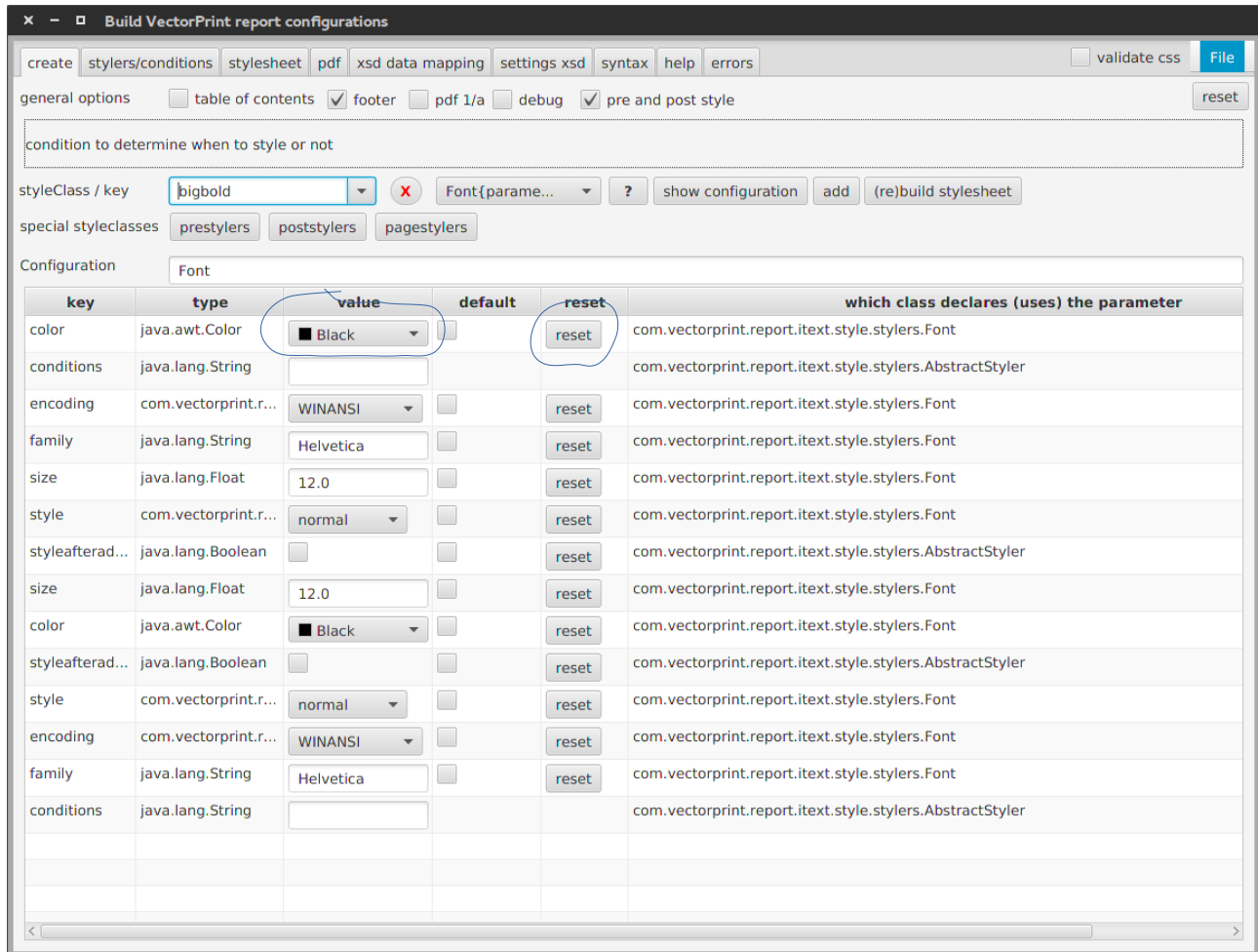
SimpleObjectProperty allows you to propagate changes to you own observable objects using the standard binding mechanism JavaFX offers and vice-versa. The example here stresses possibilities to the limit. We have an own own object that is Observable, we have a wrapper for this object that is an Observer and also extends SimpleObjectProperty. Changes to the wrapped object will be notified to the wrapper which can in turn notify the GUI. Furthermore user changes to properties (from the GUI) will be propagated to the wrapper which will then update the value of the wrapped object.

```
/**
 * This class is a SimpleObjectProperty so GUI nodes can be auto updated, it is
 * an Observer so that when the
 * encapsulated {@link Parameter} changes this class is notified and can update
 * its value, it is a ChangeListener so
 * when GUI nodes change this class is notified and can update its encapsulated
 * Parameter value.
 *
 * @author Eduard Drenth at VectorPrint.nl
 */
public class ParameterProps<T extends Serializable> extends
SimpleObjectProperty<ParameterProps<T>> implements Comparable<ParameterProps>,
ChangeListener, Observer {

    @Override
    public void changed(ObservableValue observable, Object oldValue, Object
newValue) {

    @Override
    public void update(Observable o, Object arg) {
        value = ....
        fireValueChangedEvent();
    }
}
```

In the below image the value cell will be updated through binding when the reset button is clicked. Also cells with small GUI Nodes will reflect the correct reset value.



setup observing to keep values and GUI in sync

In order to make the mechanism described above work you need to register the wrapper as observer and as value in the SimpleObjectProperty super class.

```
public ParameterProps(Parameter<T> p) {
    this.p = p;
    p.addObserver(this);
    set(this);
}
```

custom cellFactory

A CellFactory is responsible for drawing contents of a Cell. **Note** that a Cell here does not necessarily mean table cell. In the factory you return a new Cell whose updateItem method you override that will be called by from a JavaFX graphical thread. In the method you can control the look and feel of the cell (setText, setGraphics,...).

```
parameterizableCombo.setCellFactory((ListView<Parameterizable> p) -> {
    return new ListCell<Parameterizable>() {
        @Override
        protected void updateItem(Parameterizable t, boolean bln) {
            super.updateItem(t, bln);
            setText(t == null ? "" : t.getClass().getSimpleName());
            setTooltip(t != null ? ViewHelper.tip(help(t)) : null);
        }
    };
});
```



```
});
```

a custom CellValueFactory

You can use a `CellValueFactory` to yield the value of a `Cell`. This value must be observable, you may achieve this using a JavaFX wrapper class, or better create your own `ObservableValue` to take full advantage of binding possibilities. In this case the value of the `pValue` `Cell` is changed by clicking on a button in another `Cell`, this change is then propagated to the GUI using `fireValueChangedEvent()`.

```
pValue.setCellValueFactory(new
    Callback<TableColumn.CellDataFeatures<ParameterProps,
ParameterProps>, ObservableValue<ParameterProps>>() {

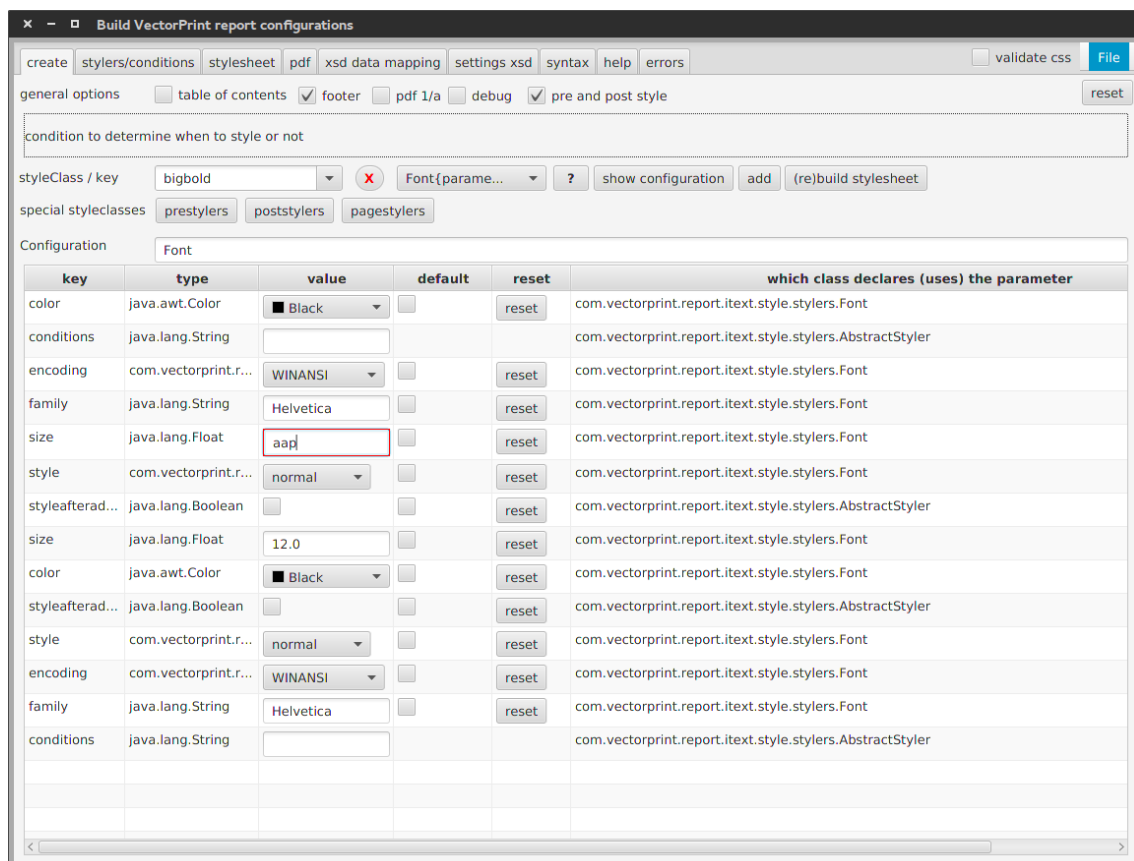
    @Override
    public ObservableValue<ParameterProps>
    call(TableColumn.CellDataFeatures<ParameterProps, ParameterProps>
param) {
        return param.getValue();
    }

});
```

a ColorPicker as content of a cell

In this example you see the use of a color picker to set the color value of an item. The item is a `ChangeListener` that will be notified when a color is picked. Also you see an example where a value is validated on every keystroke showing a red border (or some other error style) until the value is correct. **Note** the `setGraphic(null)` which is crucial to wipe out old data in the GUI!

```
pValue.setCellFactory((TableColumn<ParameterProps, ParameterProps>
param) -> {
    return new TableCell<ParameterProps, ParameterProps>() {
        @Override
        protected void updateItem(final ParameterProps item, boolean
empty) {
            setGraphic(null);
            final ColorPicker cp = new ColorPicker();
            cp.valueProperty().addListener(item);
            setGraphic(cp);
        } else {
            final TextField textField = new TextField(item.getVal());
            textField.textProperty().addListener((ObservableValue<?
extends String> observable, String oldValue, String newValue) -> {
                textField.getStyleClass().remove("error");
                try {
                    item.setValue(newValue);
                } catch (Exception e) {
                    textField.getStyleClass().add("error");
                    ViewHelper.writeStackTrace(e, error);
                }
            });
            setGraphic(textField);
        }
    }
}
```



A value is incorrect, the value field is styled accordingly. As soon as the typed value becomes

correct the “error” style is removed.

FXCollections

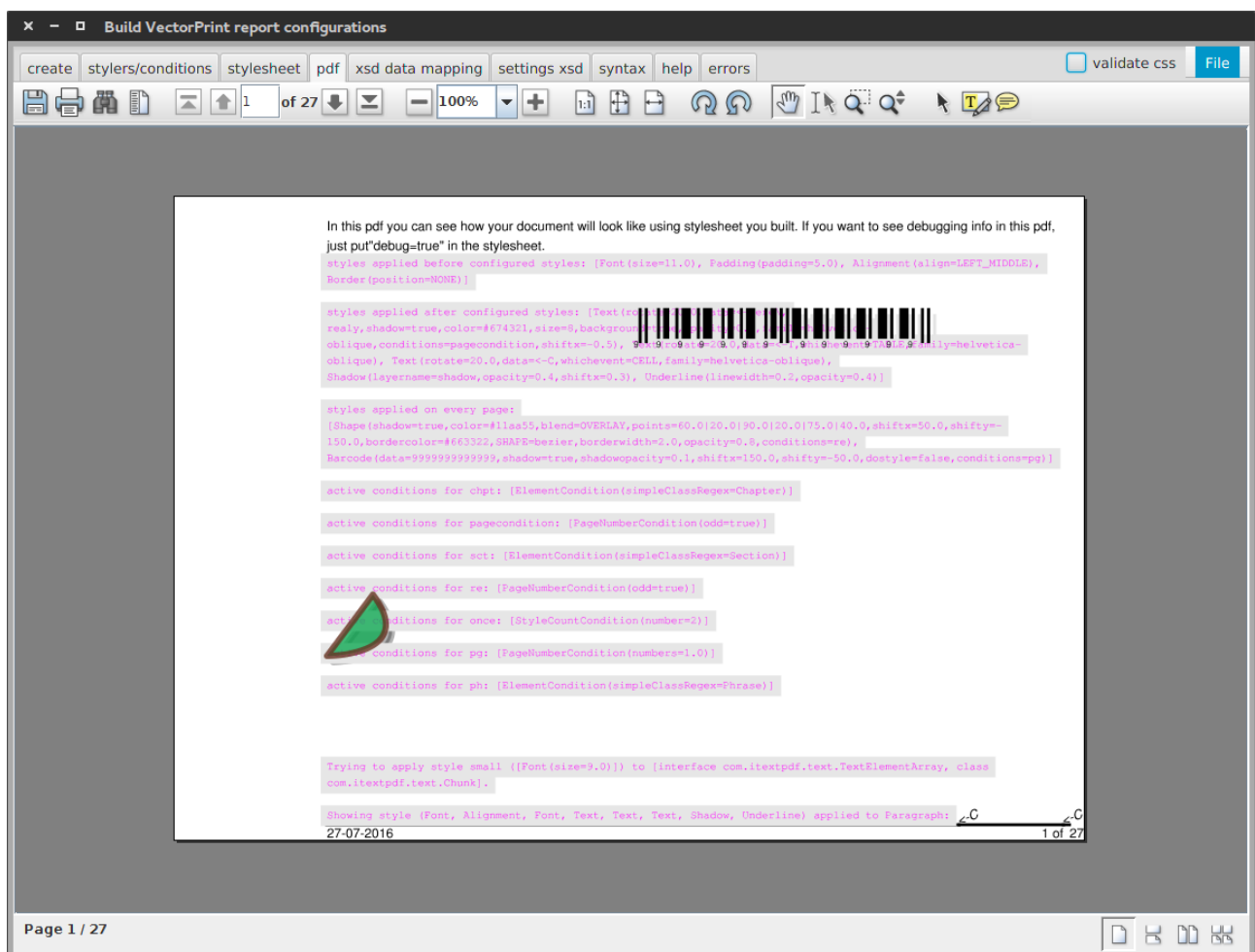
Here a regular list is used for a dropdown, the list is wrapped in an ObservableList for this. **Note** that changes to the wrapper list are not propagated to the original list none the other way around.

```
parameterizableCombo.setItems(FXCollections.observableArrayList(sorted));
```

swing integration

You can embed a complete swing app in JavaFX and the other way around through JFXPanel. **Note** that the only relation between container and contained is showing the graphics, events etc. will not be propagated to and from.

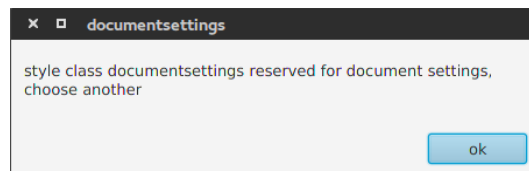
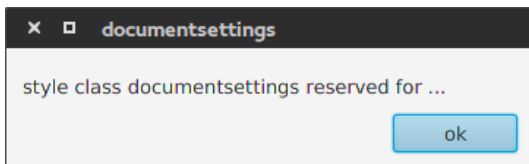
```
<javafx.embed.swing.SwingNode fx:id="pdfpane" AnchorPane.bottomAnchor="0"
AnchorPane.leftAnchor="0" AnchorPane.rightAnchor="0" AnchorPane.topAnchor="0" />
```



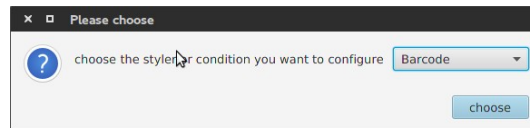
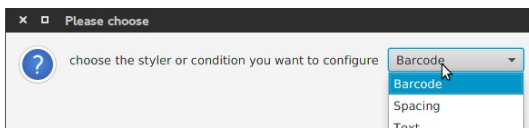
built-in dialogs

Some built-in dialogs are available, easy to use but not very flexible and customizable. Below I had to introduce a wrapper object with a toString method that shows some sensible text in the dropdown. Whereas I would prefer to be able to specify a cellFactory. Also I wanted to use one button and had to call clear() to get rid of the default ones. Last but not least I had to set the headerText to null in order to get closer to the way I wanted the ChoiceDialog to look.

```
List<ParameterizableWrapper> pw = new ArrayList<>(stylers.size());
stylers.stream().forEach((p) -> {
    pw.add(new ParameterizableWrapper(p));
});
Dialog<ParameterizableWrapper> dialog = new ChoiceDialog<>(pw.get(0), pw);
ButtonType bt = new ButtonType("choose", ButtonBar.ButtonData.OK_DONE);
dialog.getDialogPane().getButtonTypes().clear();
dialog.getDialogPane().getButtonTypes().add(bt);
dialog.setTitle("Please choose");
dialog.setHeaderText(null);
```



Images of the default dialog, one resized, the other not. Below a not so nice standard choice dialog.



FileChooser

```
FileChooser fc = new FileChooser();
fc.setTitle("add jar");
FileChooser.ExtensionFilter extensionFilter = new
FileChooser.ExtensionFilter("Jar files (*.jar)", "*.jar", "*.JAR",
"*.Jar");
fc.getExtensionFilters().add(extensionFilter);
List<File> l = fc.showOpenMultipleDialog(null);
```

packaging: javafx-maven-plugin

This maven plugin is a wrapper around the options Java has for packaging and deploying Java(FX) apps. See <http://docs.oracle.com/javafx/2/deployment/jfxpub-deployment.htm>. The good thing being you will be IDE independent and have many options for automation of deployment etc. The bad thing that for running, debugging, profiling and testing from your IDE you will have to do some extra.

You can for example use `mvn jfx:jar` to create an executable jar in `target/jfx/app`, or `jfx:native` to create an executable (installer, .deb, .rpm) in `target/jfx/native`.

```
<build>
  <plugins>
    <plugin>
      <groupId>com.zenjava</groupId>
      <artifactId>javafx-maven-plugin</artifactId>
      <version>8.5.0</version>
      <configuration>
        <mainClass>${mainClass}</mainClass>
      </configuration>
    </plugin>
  </plugins>
</build>
```

use exec-maven-plugin for running / debugging

You can use the exec-maven plugin for running, debugging and profiling. Testing I haven't looked into yet.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <version>1.5.0</version>
  <executions>
    <execution>
      <id>default-cli</id>
      <goals>
        <goal>exec</goal>
      </goals>
      <configuration>
        <executable>${java.home}/bin/java</executable>
        <arguments>
          <!--<argument>-
agentlib:jdwp=transport=dt_socket,server=y,address=8000</argument>-->
          <argument>-jar</argument>
          <argument>${basedir}/target/jfx/app/${
project.build.finalName}-jfx.jar</argument>
        </arguments>
      </configuration>
    </execution>
  </executions>
</plugin>
```