JavaFX Scene Graph Object Serialization

ELMIRA KHODABANDEHLOO

KTH Information and
Communication Technology

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

JavaFX Scene Graph Object Serialization

Elmira Khodabandehloo

Master of Science Thesis

8 October 2013

Examiner
Professor Gerald Q. Maguire Jr.

Supervisor
Håkan Andersson

School of Information and Communication Technology (ICT)
KTH Royal Institute of Technology
Stockholm, Sweden

Abstract

Data visualization is used in order to analyze and perceive patterns in data. One of the use cases of visualization is to graphically represent and compare simulation results. At Ericsson Research, a visualization platform, based on JavaFX 2 is used to visualize simulation results. Three configuration files are required in order to create an application based on the visualization tool: XML, FXML, and CSS.

The current problem is that, in order to set up a visualization application, the three configuration files must be written by hand which is a very tedious task. The purpose of this study is to reduce the amount of work which is required to construct a visualization application by providing a serialization function which makes it possible to save the layout (FXML) of the application at run-time based solely on the scene graph.

In this master's thesis, possible frameworks that might ease the implementation of a generic FXML serialization have been investigated and the most promising alternative according to a number of evaluation metrics has been identified. Then, using a design science research method, an algorithm is proposed which is capable of generic object/bean serialization to FXML based on a number of features or requirements. Finally, the implementation results are evaluated through a set of test cases. The evaluation is composed of an analysis of the serialization results & tests and a comparison of the expected result and the actual results using unit testing and test coverage measurements.

Evaluation results for each serialization function show that the results of the serialization are similar to the original files and hence the proposed algorithm provides the desired serialization functionality for the specific features of FXML needed for this platform, provided that the tests considered every aspect of the serialization functionality.

Keywords: Scene graph, Java bean, JavaFX, FXML, serialization, reflection, introspection, DOM, tree traversal.

Sammanfattning

Datavisualisering används för att analysera och uppfatta mönster i data. Ett användningsfall för visualisering är att grafiskt representera och jämföra simuleringsresultat. På Ericsson Research har en visualiseringplattform för att visualisera simuleringsresultat utvecklats som baserats på JavaFX 2. Tre konfigurationsfiler krävs för att skapa en applikation baserad på denna visualiseringsplattform: XML, FXML och CSS.

Det nuvarande problemet är att för att utveckla en ny applikation så måste de tre konfigurationsfilerna skrivas för hand vilket är kräver mycket utvecklingstid. Syftet med denna studie är att minska mängden arbete som krävs för att konstruera en visualiseringapplikation genom att tillhandahålla en serialiseringsfunktion som gör det möjligt att spara applikationens layout till en FXML-fil medan programmet exekverar enbart genom att extrahera information ur det grafiska gränsnittets scengraf.

I detta examensarbete har ett antal mjukvarubibliotek eller API: er som kan underlätta utvecklandet av en generisk FXML serialiseringsfunktion analyserats och de mest lovande alternativen enligt ett antal utvärderingsmetriker har identifierats. Med hjälp av en iterativ, design-orienterad forskningsmetod har en algoritm designats som är kapabel till att serialisera generiska Java-objekt, eller Java-bönor till FXML. Den föreslagna algoritmen har sedan utvärderats genom automatiserade mjukvarutester. Utvärderingen består av: analys av serialiseringsresultat, design av testfall, samt jämförelse av förväntade resultat och de faktiska resultaten med hjälp av enhetstest och uppmätt kodtäckning.

Utvärderingen visar att serialiseringsalgoritmen ger resultat som motsvarar de ursprungliga FXML-filerna som utformats för att verifiera olika delar av FXML standarden. Därmed anses den föreslagna serialiseringsalgoritmen uppfylla de delar av FXML-specifikationen som kravställts och beaktats i detta examensarbete.

Nyckelord: scengraf, Java-bönor, JavaFX, FXML, serialisering, reflection, introspection, DOM, träd traversering.

Acknowledgment

I would like to express my deepest gratitude to all those who provided me with great support during this master's thesis project.

A special gratitude to **Håkan Andersson**, my supervisor at Ericsson AB for his motivation, great knowledge and wisdom, valuable support and ideas, great feedbacks and helping in every aspect of this work.

Furthermore, I would like to express my sincere gratitude to my academic supervisor and examiner at KTH, **Professor Gerald Q. Maguire Jr.** for his great support by providing me with extremely helpful feedback and suggestions through the whole period of this master's thesis project.

I also take the opportunity to greatly appreciate **Claes Tidestav**, my manager at Ericsson Research for helping and giving me this great opportunity.

Even though my gratitude can not be expressed by words to them, I would like to express my appreciation to my mother, father and brother in Iran for their unconditional support and love throughout my life.

Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BFS	Breadth-First Traversal
CSS	Cascading Style Sheets
DFS	Depth-First Traversal
DOM	Document Object Model
EJB	Enterprise Java Beans
FXML	JavaFX eXtensible Markup Language
GSM	Global System for Mobile Communications
GUI	Graphical Uset Interface
ITU	International Telecommunication Union
JAXB	Java Architecture for XML Binding
JAXP	Java API for XML Processing
JDOM	Java Document Object Model
JINI	Java Intelligent Network Infrastructure
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LTE	Long Term Evolution
RMI	Remote Method Invocation
SAX	Simple API for XML
SE	Standard Edition
SQL	Structured Query Language
VRML	Virtual Reality Modeling Language
W-CDMA	Wideband Code Division Multiple Access
WYS/WYG	Whay You See/ What You Get
W3C	World Wide Web Consortium
XML	eXtensible Markup Language
1G	First Generation (mobile telephony)
2G	Second Generation (mobile telephony)
3G	Third Genetration (mobile telephony)
3GPP	Third Generation Partnership Project
4G	Fourth Generation (mobile telephony)

1 Introduction

This chapter first provides an overview of the research area. This is followed by the definition of the problem. Next the research goal is described in terms of a set of sub-goals. The scope of the thesis is specified. This chapter ends with a description of the structure of the thesis enumerating the purpose of the different chapters of this thesis.

1.1 General introduction to the area

Most software applications rely on mechanisms for persistent data storage. A basic requirement for this persistent storage is for an application to take an object and write it in a format which is suitable for storage and transmission. Later the application will reconstruct a copy of the object (from the stored form of the object) which is equivalent to the original object [1, 2].

In computer science, this conversion process from a data structure to a sequence of bytes is called serialization or marshaling [3]. Serializaiton is widely used with component-based softwares [2] and can be used to create remote instances of objects [4]. During the serialization process, the state of an object is encoded into a format which is suitable for storage in a file or transmission over a network [5]. The reverse process is called deserialization or unmarshaling, which implies recreation of the object from its serialized representation [5].

The structured data can be serialized either into binary or ASCII (American Standard Code for Information Interchange) format. There are various object serialization libraries that support different programming languages. When it comes to Java object serialization, a persistent mechanism is provided in the standard Java package which can serialize an object and all objects referenced by that object into a binary format [6]. This standard mechanism is one of the underlying concepts on which various high-level Java technologies such as Java RMI (Remote Method Invocation), EJB (Enterprise JavaBeans), and JINI (Java Intelligent Network Infrastructure) are built [3].

Despite the advantages of standard Java object serialization, multiple drawbacks exist for it both from a qualitative and a quantitative point of view. Firstly, it is not feasible for other programming languages to make use of the binary data due to the lack of libraries which would enable them to read the data. Secondly, the size of a binary representation of the object is fairly large compared to other formats. Finally, the binary serialization format is not easily readable by humans [6].

Sometimes it is desirable to have a machine independent text-based (typically ASCII representation) of Java objects [6]. Additionally, studies have shown that the size of an ASCII-based representation of Java objects is smaller than the correspondent binary one (as produced by the standard Java object serialization) [6]. Today, there are several different widely used text-based data representations, such as XML and JSON. As a result, many Java object serialization libraries have been developed to serialize Java objects into these formats. XML is a markup language which is platform independent and also an alternative for translating Java objects into a human-readable format. XML is highly extensible and there exist many XML writers and parsers. For these reasons, XML has been widely adopted as a standard format for representation of data [6]. These characteristics also make XML an appropriate candidate for integration of different software systems running on various platforms [7].

This master's thesis concerns XML serialization and focuses on a particular XML-based markup language called FXML, which was specifically created to represent the layout or user interface of JavaFX applications [8].

1.2 Problem definition

Different mobile telecommunication technologies have emerged and evolved all around the world. Some of these technologies have become ubiquitous. They have evolved through multiple so-called generations, beginning with the first generation (1G) which is based upon analog mobile radio – this was introduced in the 1980s. These 1G systems were followed by the second generation (2G). They were the first digital mobile radio systems. One of the more important of these 2G standards was the GSM standard. GSM achieved very wide adoption around the world. Today, the latest evolution of the digital mobile communication systems family defined by the Third Generation Partnership (3GPP) is called long term evolution (LTE) and was introduced in 3GPP's release 8 [9]. 3GPP has continued to evolve their standards and today LTE-Advanced is considered by ITU (International Telecommunication Union) to be a fourth generation (4G) standard.

Ericsson Research conducts development and research into these technologies by modeling and simulating both complete systems and the radio network elements. One important area which is studied is the radio network algorithms used in radio network simulators. Research studies and simulation results from Ericsson Research are presented either internally within Ericsson or to external parties (such as the 3GPP standardization organization or customers - typically mobile network operators). However, to present complex simulation results which might consist of huge amounts of data, a visualization tool is needed in order to make it easier to analyze and perceive patterns in the data. A visualization platform has been developed at Ericsson based on JavaFX 2.x to address this need. It is a modern Java environment that eases deployment and maintenance of data-driven business and enterprise client applications by separation of the application logic and the application user interface [10].

This visualization platform provides an interactive graphical user interface (GUI) which is composed of different visual components. These components are basically heavy weight JavaFX controls (such as pie chart, line chart, map, scatter chart, bubble chart, and so forth). By selecting a particular scenario and case (which corresponds to a certain set of data), the simulation results are visualized as interactive charts.

The input data to the visualization platform is provided by a SQL server database. There are three configuration files which are required when creating an application on top of the Ericsson visualization tool. These three configuration files are associated with the boxes in the top row of Figure 1-1.

Figure 1-1: Visualization platform's application definition [11].

The first configuration file represents the application model which is encoded in an XML file. The application model primarily contains meta-data descriptions of the SQL relational database tables. These are required by the platform in order to allow bindings between the database, the XML-file, and the FXML objects. In other words, the main purpose of the application model is to specify meta-data descriptions of the SQL relational-database, including primary and foreign key constraints, in order to build a client-side object representation of the remote SQL database.

The second configuration file is an FXML file. FXML is a JavaFX extension of the XML markup language and deals with the *layout* of the user interface to an application. More specifically, an FXML file comprises a hierarchical structure of the GUI elements which represents JavaFX controls and makes it possible to define a complete user interface, as well as defining a means for the handling of application events. FXML decouples the user interface from the application logic, thus making FXML a good substitution for procedural coding of the user interface by developers [10].

The last configuration file is a JavaFX Cascading Style Sheets (CSS) that contains the style definitions which can be used by the JavaFX controls or objects in the scene (represented as a graph). The CSS file handles branding, color, font, and the appearance of the JavaFX application [12].

Development of a new application based on the visualization platform requires creation of these three configuration files (FXML, XML, and CSS) manually from scratch. This can be a problem for a large company with huge data sets of simulation results, such as Ericsson. The need to manually generate these three files motivated the improvement of the existing visualization platform, in order to promote usability and to reduce the effort required by users of the visualization platform. The goal is to improve the platform by automatically generating the required configuration files and providing a new "save" function for the visualization platform based on JavaFX.

This master's thesis concentrates on the FXML-related (i.e., the layout) part of an application which is to be developed based on the visualization tool. Loading the FXML file elements that comprise JavaFX scene graph's objects is already supported in the Java SE framework [13]. However, since every application addresses different needs and hence the requirements on the GUI elements change, it is a tedious task to develop an application layout FXML from scratch every time a new application is to be developed. To avoid this very labor intensive approach, the serialization process should automatically extract the FXML representation of the GUI elements from the JavaFX scene graph. Unfortunately, there is no generic or widely accepted solution to do this. This leads directly to the goals described in the following section.

1.3 Goals

This master's thesis project is conducted in cooperation with Ericsson Research, in particular the organization known as "Wireless Access Networks". It is of great interest to Ericsson Research to save the layout of the visualization platform's applications to persistent storage after they have been created by a designer at run-time. To achieve this goal, it is necessary to generate an FXML representation of the JavaFX application scene graph. Hence, this general goal is divided into more specific sub-goals:

- The first sub-goal of this master's thesis is to investigate possible frameworks that may facilitate the implementation of an algorithm for generic FXML serialization. As part of this process, it is required to identify the benefits and limitations (or drawbacks) of each alternative.
- The second sub-goal is to select the most promising alternative identified in the previous sub-goal based upon a number of evaluation metrics. This requires proposing and designing a new algorithm or solution which is capable of generic Java bean serialization to FXML.
- The last sub-goal is to implement the algorithm in Java/JavaFX according to the Ericsson code conventions and verify (and evaluate) the results through suitable series of test cases based on the FXML specification.

1.4 Scope

The theoretical aspects of this thesis will include the following topics:

- Different features of FXML, in particular, its syntax and semantics, and how it is used in JavaFX. The following functionality of FXML are expected to be supported in this thesis: traversing the scene graph, transformation of DOM tree to a valid FXML, detecting the changed properties of the scene graph objects, managing the import statements of FXML files, primitive or atomic properties, support for lists, particularly ObservableList of FXCollections collections, support for immutable objects, support for static properties, support for enumerations and fx:controller.
- The rest of the FXML features are out of the scope of this thesis, since they are not really needed for the requested functionality. These features are: support for fx:define, fx:id, event handlers, location resolution, variable resolution, escape sequences, expression bindings, preserving fx:include relations, support for fx:root, fx:script, fx:copy, fx:reference, and support for ObservableMap.
- Previous research that is related to serialization: in particular, XML serialization which is specifically used in this task.
- Identifying different alternatives for generic FXML serialization as well as determining benefits, restrictions, or drawbacks of each alternative.
- Selecting the most promising serialization alternative according to the evaluation criteria and finally designing a solution or algorithm for generic JavaFX object graph to FXML serialization.

The practical design, implementation, and evaluation aspect of this thesis project include:

- Designing and implementing in Java/JavaFX a solution based upon the information in the theoretical portion of this thesis.

- Verifying and evaluating the results of the implemented serialization functionality by means of test cases. These tests should ensure that the serialized FXML files are in compliance with the FXML specification.

1.5 Structure of the thesis

Chapter 1 provided an overview of this master's thesis, including identifying the topic and describing the problem. This chapter also stated the aim, goal (and sub-goals), and scope of this project. **Chapter 2** provides the required background to comprehend the research undertaken in this thesis project. It describes different elements of the FXML specification and how they are used in JavaFX with regard to FXML syntax and semantics. This is followed by a description of the different APIs and libraries that might be used to realize the serialization process. The chapter ends with a description of some related XML serialization research. **Chapter 3** describes the method used to solve the problem and evaluate the result. **Chapter 4** describes the design and functional requirements. The proposed solution is presented and discussed in **chapter 5**. This chapter comprises both API selections to come to a design decision as well as considerations concerning the proposed algorithm. **Chapter 6** presents the results of the unit tests used to verify the requirements and algorithm design. Evaluation results are discussed and analyzed and suggestions for future work are given in **Chapter 7** which concludes this thesis.

2 Background

This chapter introduces the concepts, application APIs, and libraries that are necessary to understand the rest of this thesis. First, a brief description of the JavaFX API and FXML language is provided. It should be mentioned that the emphasis in Section 2.1 is to introduce the different features of the FXML markup language specification and how they are applied in JavaFX, rather than providing a complete introduction to the JavaFX API.

Only the sub-set of the FXML specification that is within the scope of this master's thesis will be considered. The other FXML features will simply be mentioned in order to give the reader a comprehensive overview of the entire FXML markup. However, some known issues (particularly drawbacks) of these FXML features will be briefly mentioned.

The next chapter introduces several APIs and libraries that might facilitate the implementation of the JavaFX serialization into FXML. These APIs and libraries constitute potential candidates that might contribute to the design decision. These alternatives also give the reader insight into the design decisions that will be taken in the Design chapter. This chapter ends with a review of related research in serialization, in particular XML serialization. However, at the time of the writing of this thesis there exists no generic FXML-related serialization.

2.1 An overview of JavaFX and FXML

To have a holistic view of FXML and its utilization in JavaFX, it is necessary to review the foundations of JavaFX.

In May 2007, Sun Microsystems announced a new member of the Java family: JavaFX. The primary purpose of the JavaFX platform was to accelerate and simplify the development and deployment of rich client applications. Josh Marinacci, a software engineer at Sun stated that: "JavaFX is sort of a code word for reinventing client Java and fixing the sins of the past" [8].

Currently, Oracle's JavaFX 2.0 is regarded as the premier modern client platform for rich enterprise applications. JavaFX 2.0 is considered as the next evolutionary stage of Java since it facilitates both development and deployment of "data driven business and enterprise client applications" [10].

2.1.1 JavaFX features and benefits

Table 2-1 describes the main JavaFX features, as well as their corresponding benefits. Figure 2-1 illustrates a visualization application developed using both JavaFX and FXML (as the user interface).

Among all of the JavaFX features, FXML is the core concept which this thesis project will concentrate on, thus this chapter will enumerate the FXML markup language specification and how FXML markup acts in conjunction with JavaFX.

Table 2-1: The main JavaFX features and their benefits

Java APIs for JavaFX	Starting with JavaFX 2 version, JavaFX applications can be developed completely in Java. This feature has made JavaFX a ubiquitous platform all over the world. According to Oracle's statistics, more than nine million developers make use of the JavaFX platform to develop rich cross-platform client applications. Since JavaFX is built on Java technology, developers can take the advantage of the underlying features of the core Java language such as multi-threading, generics, annotations, etc [10]. However, it is not mandatory to write JavaFX code using only the Java language. JavaFX offers a highly flexible means for application development in other popular Java virtual machine (JVM) based scripting languages, such as Groovy, JRuby, and Scala. Developers can take advantage of features of each of these languages to meet the requirements of the application which is to be developed [8].
FXML	JavaFX has its own definition or declaration language written in XML. This language, called FXML, is scriptable and easy-to-learn. FXML is used to construct the user interface of JavaFX applications. FXML is very useful as it decouples the application logic from the user interface [8,10]. FXML not only provides powerful user interfaces to JavaFX applications, but can also be used to specify the behavior and interactions with the user interface by exploiting the scripting features that are embedded within an FXML file [8].
Web components	It is possible to embed JavaFX applications into web components. JavaFX also allows access and manipulation using a document object model (DOM) from within the Java code [8].
Powerful properties model	JavaFX supports new designs and implementations for Java bean properties using a powerful properties model. It also introduces new collections, such as ObservableList, ObservableMap, and Sequence.

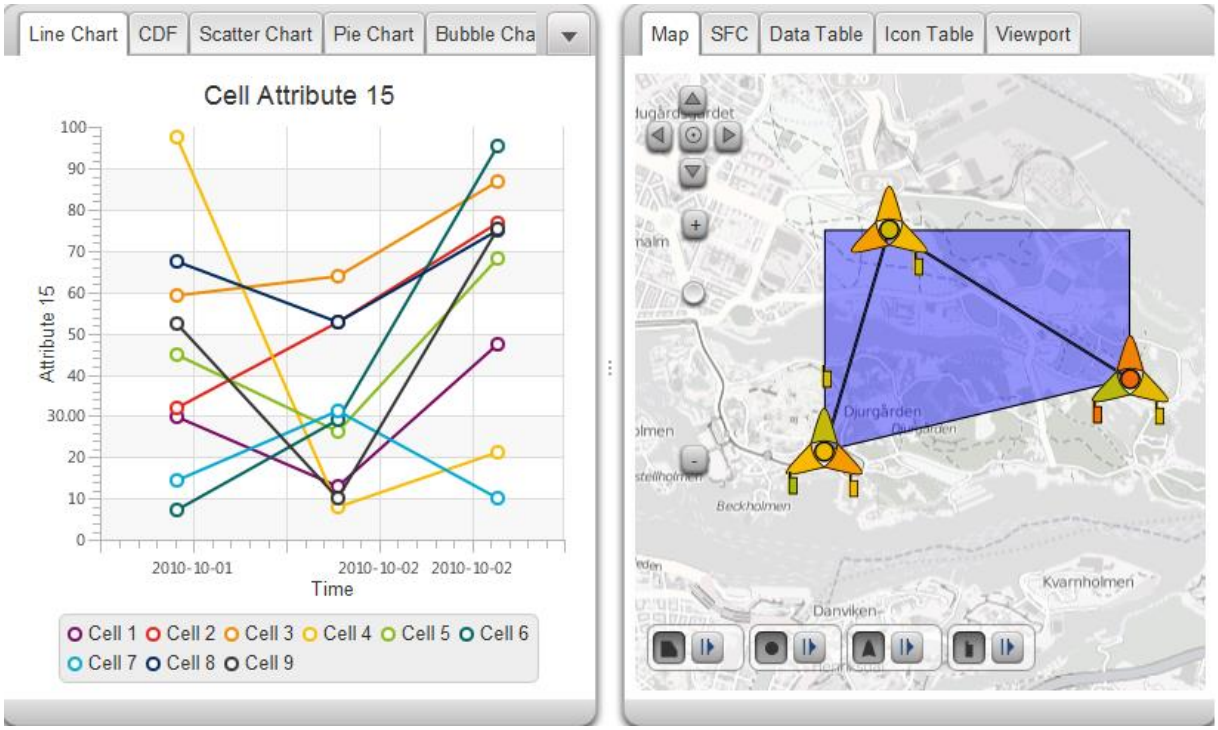


Figure 2-1: Data visualization using a mix of FXML and JavaFX [11].

2.1.2 FXML specification

FXML provides a *declarative* approach to construct a user interface for a JavaFX application [8]. Just like any other markup language, FXML is a hierarchical structure, composed of processing instructions, elements, attributes, text values, etc. There is a correspondence between the components of the FXML document and the JavaFX objects in the scene graph. To be more specific, FXML *elements* are mapped to JavaFX *classes* and FXML attributes are mapped to bean properties [14].

In order to get an idea of a scene graph structure, the scene graph concept will be decribed. A scene graph is a tree data structure that represents the hierarchical structure of items (typically called nodes). These nodes constitute the JavaFX application user interface [15]. The scene graph can be generated programmatically by Java code. However, JavaFX application developers substitute the Java code for FXML markup in order to construct a GUI. In this case, the scene graph is populated when the root of the FXML file is loaded [8]. Figure 2-2 illustrates a simple scene graph composed of various objects that represent JavaFX user interface (UI) nodes. The starting point of every scene graph is the root object which encompasses the whole scene graph.

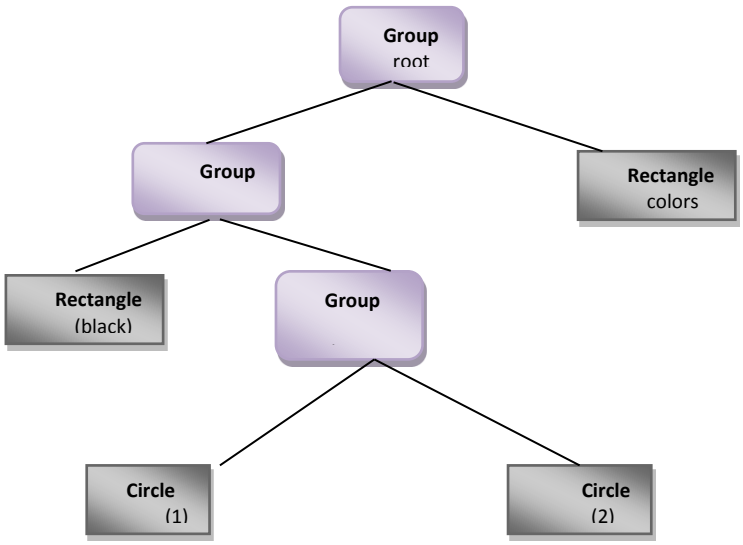


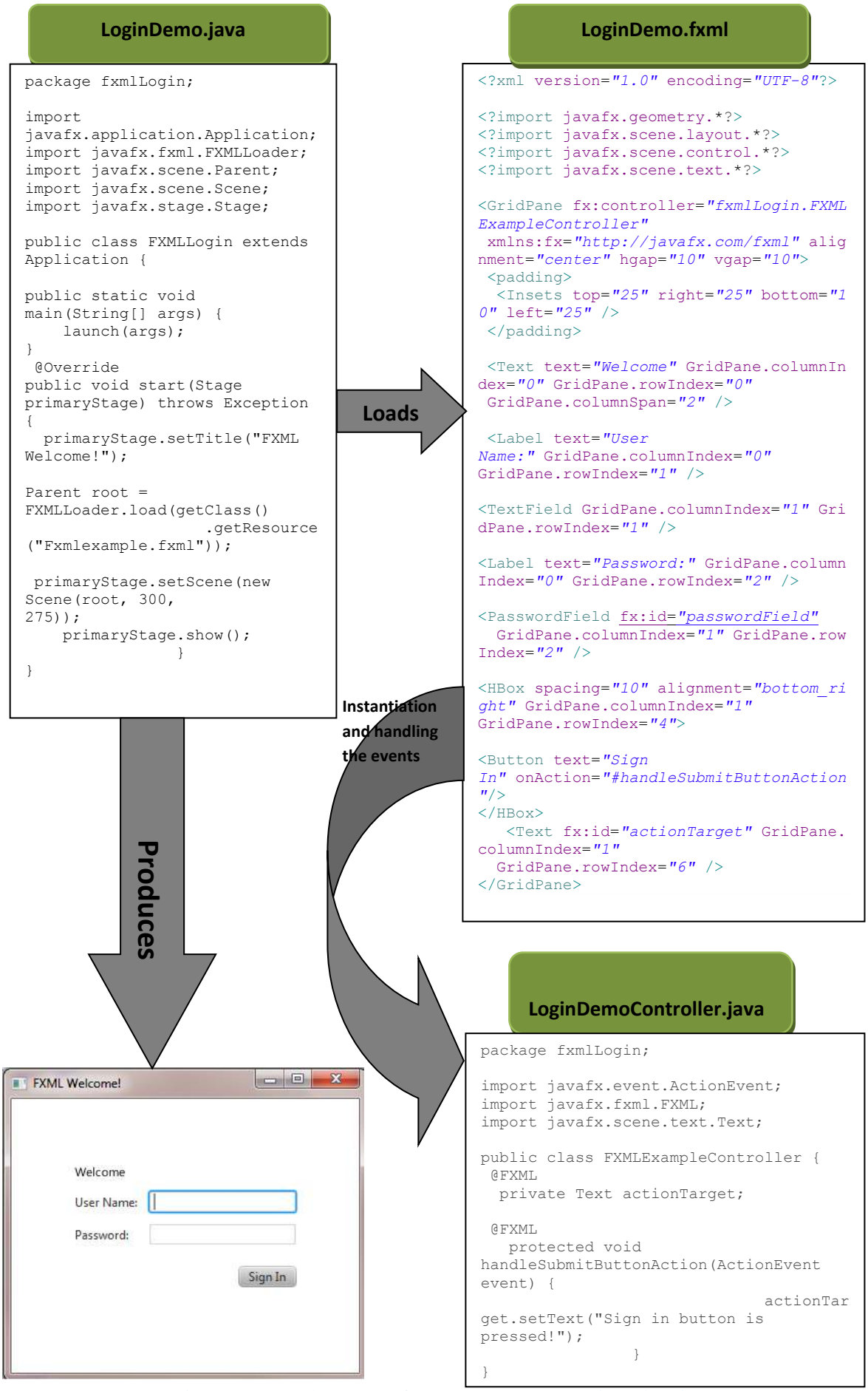
Figure 2-2: JavaFX scene graph example [16].

As depicted in Figure 2-3, the content of the FXML file is loaded into the scene graph by invoking the load method of the FXMLLoader class and then the UI of the JavaFX application is rendered.

As mentioned previously, the specifications that are in the scope of this master's thesis are described in more detailed in this section. Readers are recommended to look at the Oracle "Introduction to FXML" tutorial to get a more comprehensive insight into the FXML language syntax and semantics [13].

An FXML file is a set of elements and attributes along with attribute values. There are two approaches to set an attribute for a particular element in any XML document. It can be either directly specified in its corresponding element tag or it could be a nested child element [17]. The former is the preferred format that will be adopted for this thesis project. Hence, the core functionality is provided, mostly concerns FXML elements rather than attributes. This approach increases the readability of the generated FXML.

In an FXML document, there are different implications for XML elements. An XML element could be the representation of a class instance, a class instance property, a static property, a define block, or a script code block [13]. The first three alternatives are described in more detail, while the last two are out of the scope of this thesis.




```
package fxmlLogin;

import
javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class FXMLLogin extends
Application {

public static void
main(String[] args) {
    launch(args);
}
 @Override
public void start(Stage
primaryStage) throws Exception
{
  primaryStage.setTitle("FXML
Welcome!");

Parent root =
FXMLLoader.load(getClass()
                    .getResource
("Fxmlexample.fxml"));

 primaryStage.setScene(new
Scene(root, 300,
275));
    primaryStage.show();
                }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.text.*?>

<GridPane fx:controller="fxmlLogin.FXML
ExampleController"
 xmlns:fx="http://javafx.com/fxml" alig
nment="center" hgap="10" vgap="10">
 <padding>
  <Insets top="25" right="25" bottom="1
0" left="25" />
 </padding>

 <Text text="Welcome" GridPane.columnIn
dex="0" GridPane.rowIndex="0"
 GridPane.columnSpan="2" />

 <Label text="User
Name:" GridPane.columnIndex="0"
GridPane.rowIndex="1" />

<TextField GridPane.columnIndex="1" Gri
dPane.rowIndex="1" />

<Label text="Password:" GridPane.column
Index="0" GridPane.rowIndex="2" />

<PasswordField fx:id="passwordField"
  GridPane.columnIndex="1" GridPane.row
Index="2" />

<HBox spacing="10" alignment="bottom_ri
ght" GridPane.columnIndex="1"
GridPane.rowIndex="4">

<Button text="Sign
In" onAction="#handleSubmitButtonAction
"/>
</HBox>
   <Text fx:id="actionTarget" GridPane.
columnIndex="1"
  GridPane.rowIndex="6" />
</GridPane>
```

```
package fxmlLogin;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.text.Text;

public class FXMLExampleController {
 @FXML
  private Text actionTarget;

 @FXML
   protected void
handleSubmitButtonAction(ActionEvent
event) {
                                actionTar
get.setText("Sign in button is
pressed!");
                }
}
```

Figure 2-3: Diagram of a JavaFX FXML Application.

2.1.2.1 Class instance declaration

When it comes to declaring a class instance in FXML, there exist multiple approaches. However, the most typical approach (which is also utilized in this thesis) is to define an FXML element having the same name as the JavaFX class. For instance, the markup shown in Figure 2-4 simply creates an instance of the Button class and sets the button instance's text property to the value "Press button". It should be noted that the import processing instruction should precede the element tag and it specifies the fully qualified name of the corresponding class [13].

```
<?import javafx.scene.control.Button?>

<Button text="Press button"/>
```

Figure 2-4: Declaring a new instance of the Button class.

There are other means of instantiating a class in FXML, such as <fx:reference> which creates a reference to a previously defined object, <fx:copy> constructs an instance of a class by copying a predefined value, and <fx:include> creates a class instance based on the content of another FXML file whose URL is specified in the "source" attribute of the tag element [13]0.

These alternative approaches are out of the scope of this thesis and will not be investigated in more detail. It should be noted that there are some issues associated with <fx:reference> and <fx:copy> as in a sense they are one-way functions; which means that when it comes to serialization, we cannot store the current state in FXML file by only examining the scene graph. However, the generated FXML and the original scene graph will have the same functionality and they are equivalent in terms of rendered GUI. While they both represent the same scene graph, there will be a problem in verification and testing because these two FXML files would be textually different.

In the FXML document presented in Figure 2-4, once the FXML loader reaches the Button element which is a bean type, it creates an instance of a Button based on the FXML loader's internal mechanisms [13] and then adds this instance to the scene graph. However, there are some cases in which the object to be constructed does *not* conform to the bean conventions (for example, if the class does not have any default constructor [13]). To handle such situations, some other approaches should be used to instantiate an object. They will be explained below.

- **fx:value**: This attribute is used to instantiate an object of String type, as well as each primitive wrapper type (such as Double, Integer, Float, Boolean, etc.) as depicted in Figure 2-5. Instances of the mentioned classes do not involve any default constructor, but they do have a static `valueOf()` method by which the instance value is set [13].

```
<String fx:value="Hello!"/>
<Boolean fx:value="true"/>
<Double fx:value="2.0"/>
```

Figure 2-5 : Atomic properties instantiation using fx:value attribute.

- **fx:factory**: This attribute is the second method to construct objects that do not follow the bean conventions and their classes do not have a default constructor. In this case, a static factory method is used which has no arguments and is responsible for manufacturing a class instance. The value of the fx:factory attribute indicates the name of the factory method [13]. The markup in Figure 2-6 shows how an observable array

list instance which is one of the JavaFX specific collections, can be constructed and populated with three string values by using the fx:factory attribute.

```
<FXCollections fx:factory="observableArrayList">
                <String fx:value="One"/>
                <String fx:value="Two"/>
                <String fx:value="Three"/>
</FXCollections>
```

Figure 2-6: Object instantiation using the fx:factory attribute.

- **Builders**: Builders are another means of manufacturing objects that do not conform to the bean conventions. Immutable objects are examples of such objects [13]. The state of these objects cannot be modified after construction. As they are immutable they do not have any setter methods [16]. Immutable objects are always preferable to mutable objects when it comes to dealing with synchronization problems [18].

The mechanism that JavaFX uses to create instances of immutable objects takes advantage of a mutable helper class called Builder. Builders are constructed using the BuilderFactory interface. Instances of immutable types are constructed by invoking the `build()` method of the corresponding builder class. Image and Color are examples of immutable classes in JavaFX [13]. Figure 2-7 illustrates a Color instance which is constructed based on the set values for three properties (red, blue, and green) by invoking the `build()` method of the ColorBuilder class.

```
<Color>
     <red>
        <Double fx:value="0.5" />
     </red>
     <blue>
         <Double fx:value="1.0" />
     </blue>
     <green>
        <Double fx:value="1.0" />
    </green>
</Color>
```

Figure 2-7: Using a builder to construct instances of immutable types such as Color.

2.1.2.2 Class instance property declaration

The preferred approach in this research is to treat the Java bean properties as nested child elements rather than element attributes. The motivation is to increase the readability of the constructed FXML as well as simplifying the algorithm implementation. The property element which starts with a lowercase letter is the representative of a property setter, a read-only list property, or a read-only map property [13]. These are further described as:

- **Property setter**: An FXML element can be used to set a value for a specific property. In this case, the element content is either the representation of an instance of a nested class element or a text node [13]. Figure 2-8 depicts a property element (text property) which is used to set the value of the text property of the Button instance.

```
<?import javafx.scene.control.Button?>

<Button>
    <text>Press button</text>
</Button>
```

Figure 2-8: Setting object/bean properties using property elements.

- **Read-only list property**: A read-only list property refers to a property, whose getter returns a list and it has no corresponding setter [13].The children property element in Figure 2-9 is an example of a read-only list property. When the `getChildren()` method is invoked on the VBox root node, a list instance is returned. Every sub-element of the children property is then added to this list when the FXML is parsed.

```
<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.String?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.VBox?>

<VBox xmlns:fx="http://javafx.com/fxml">
   <children>
      <Label>
         <text>
            <String fx:value="Welcome!" />
         </text>
      </Label>
      <Button>
         <text>
            <String fx:value="Press button:" />
         </text>
      </Button>
   </children>
</VBox>
```

Figure 2-9: Adding children to the read-only list property “children” of the VBox instance.

- **Read-only map property**: A read-only map property is a bean property whose getter returns an instance of the Map and it has no setter method [13].

2.1.2.3 Static properties

Static properties might also be a representation of an FXML element and they are typically properties of the parent container of the control they are nested in, but not the control itself [13]. Static properties correspond to invoking the static getter methods of the parent container class on the contained node that affects the visual representation of the contained node within the parent container, i.e. the state is set and kept within the parent container. Static properties are further elaborated on in section 5.2.5.

2.2 Investigation of different APIs

This section investigates a number of different APIs and libraries that are candidates for being used in the design of a generic FXML serialization algorithm.

2.2.1 The Java Reflection API

Reflection is an important feature of the Java language. Although in some sense, reflection is regarded as a backdoor feature in Java [19]. The starting point for the Java Reflection API

is the java.lang.reflect package which contains various classes and interfaces that reflect information about the structure and internals of Java classes [20].

Reflection is primarily used to inspect the run-time behavior of a class, to specify the structure of a Java object in order to reveal its fields, methods, and constructors. It is also possible to modify the value of fields, invoke methods dynamically, and instantiate new objects of the class using reflection. It is worth mentioning that all of these operations are within the scope of security manager restrictions [19].

In order to exploit the power of the Reflection API, the first step is to access a Class object [21] and then invoke the desired methods on this Class object in order to inspect the class internals or to manipulate the class or class instance at run-time.

It should be mentioned that the Java Reflection API consists of many classes which provide different methods or functions. However, addressing all of these methods is outside the scope of this thesis. Instead, the focus is on the methods most relevant to the task at hand. Table 2-2, Table 2-3, and Table 2-4 illustrate methods that can be used to get the different features of a class using reflection.

Table 2-2: Methods of Class for field introspection [22].

Method	Description
`Field getField( String name)`	Returns a *Field* object that represents the specified public member field of the class or interface represented by this *Class* object.
`Field[] getField`	Returns an array of *Field* objects that represents all the accessible public fields of the class or interface represented by this *Class* object.
`Field getDeclaredField( String name)`	Returns a *Field* object that represents the specified declared field of the class or interface represented by this *Class* object.
`Field[] getDeclaredFields()`	Returns an array of *Field* objects that represents each field declared by the class or interface represented by this *Class* object.

Table 2-3: Methods of Class for constructor introspection [22].

Method	Description
`Constructor getConstructor(Class[] parameterTypes)`	Returns the public *constructor* with specified argument types if one is supported by the target class.
`Constructor getDeclaredConstructor(Class[] parameterTypes)`	Returns the constructor with specified argument types if one is supported by the target class.
`Constructor[]getConstructors()`	Returns an array containing all of the public constructors supported by the target class.
`Constructor[]getDeclaredConstructors()`	Returns an array containing all of the constructors supported by the target class.

Table 2-4: Methods of Class for method introspection [22].

Method	Description
`Method getMethod (String name, Class[] parametertypes)`	Returns a *Method* object that represents a public method (either declared or inherited) of the target *Class* object with the signature specified by the second parameters.
`Method[] getMethods()`	Returns an array of *Method* objects that represents all the public methods (either declared or inherited) supported by the target *Class* object.
`Method getDeclaredMethod(String name, Class[] parametertypes)`	Returns a *Method* object that represents a declared method of the target *Class* object with the signature specified by the second parameters.
`Method[]getDeclaredMethods()`	Returns an array of Method objects thatrepresent all of the methods declared by the target *Class* object.

To better understand the reflection mechanism, Figure 2-10 shows some of the capabilities of the Reflection API including:

1. Constructing a Class object.
2. Getting a particular method of the Class object.
3. Invoking the method on a target object.
4. Getting the fully-qualified name of a class.

```
private static void setObjectColor(Object obj, Color color) {

    Class clz= obj.class;  1

                                2
     try{
        Method method= clz .getMethod ("setColor", new Class[](Color.class));

        Method .invoke (obj, new Object[] (color));
        }
                3
Catch(NoSuchMethodException ex){
         throw new IllegalArgumentException{
         clz .getName() + "does not support method setColor(color)");
                                                    }
               4                        }
}
```

Figure 2-10: Sample reflection code [22].

Despite the many functions and benefits of the Java Reflection API there are some drawbacks, such as:

- Reflection reveals internals of classes that are not exposed in non-reflective mode (such as private methods and fields).
- Security-related limitations are applied to reflection by the security manager at run-time.
- The reflection mechanism's performance is lower than a non-reflective one due to the nature of reflection which prevents the JVM from performing some optimizations [23].

2.2.2 The Java Introspection API

Introspection is the process by which Java bean features (including properties, methods, and events) are exposed and examined at run-time. The starting point for the Introspection API is the java.beans package. Introspection uses reflection to extract details of a bean class [24].

To get information about a particular bean class, the `getBeanInfo()` method on the Introspector class (included in java.beans package) can be invoked. This method provides information regarding a bean, which can subsequently be used to query different features of a bean (such as the bean's properties, methods, and events) which in turn are provided by related descriptor classes [25] depicted in Figure 2-11.

Every feature descriptor class reflects explicit information regarding a particular bean. It is just the matter of selecting a suitable descriptor to meet the requirements. One example of a descriptor which might be applied in the FXML serialization algorithm, is the PropertyDescriptor that exposes information (such as type, name, and read and write methods) of each bean property.

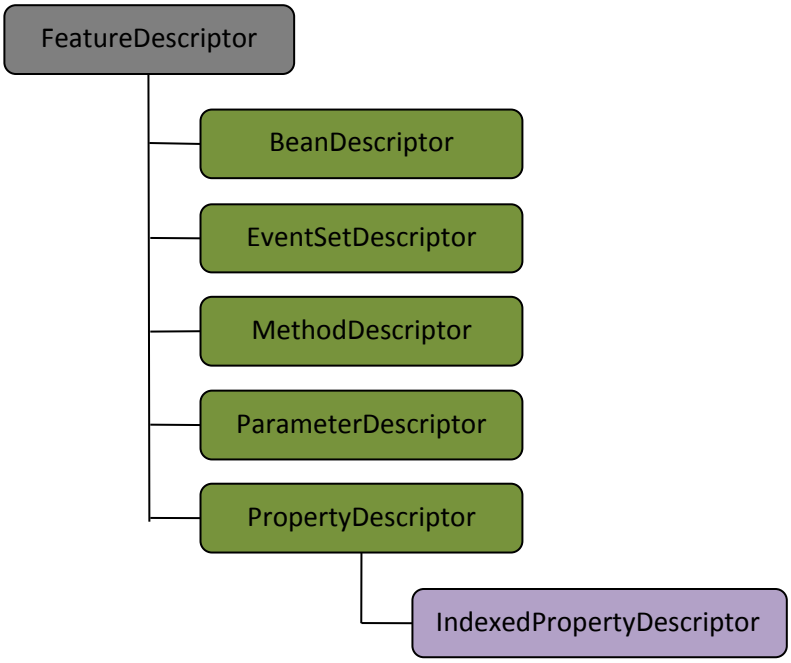


Figure 2-11: Hierarchy of the FeatureDescriptor classes [25].

2.2.3 JAXP Java API for XML Processing and serialization

In Java, there are a number of different APIs for processing XML documents. Among these, the two main APIs are Simple API for XML (SAX) and Document Object Model (DOM). Together, these APIs constitute JAXP, which is a standard part of Java1.4 and later versions of Java [26]0. SAX and DOM APIs are briefly described below.

SAX This API is typically referred to as "the gold standard of XML APIs" [26]. It is an event-driven API [27]. It does not model the document, which makes it efficient with respect to speed and the memory space, while at the same time, the lack of document model could be a disadvantage of SAX. SAX is used to model the parser and the application that gets data from the parser [26]. SAX provides sequential access to the document [27] and it suits situations in which it is not necessary to load or access the whole document in memory, for example simply to access a particular element in a document [26].

DOM DOM is a standard interface introduced by W3C [27]. DOM is platform and language *independent*. Unlike SAX, which is merely a parser, DOM makes it possible to write into a document as well. DOM models the whole document as a tree and assigns a Document object to the tree which can be queried and updated [26]. For this reason the DOM API is usually regarded as a convenient API since it is quite straightforward to access any element or part of a document at any time. A drawback of DOM, that should be highlighted, is that it is not possible to use DOM until the whole XML document is loaded into memory. This is a major issue with respect to execution time and memory when parsing huge XML documents. For smaller documents, DOM is a suitable alternative to use with JAXP [26].

As mentioned previously, both SAX and DOM are supported by JAXP. It is up to the programmers to decide which one to use based upon their requirements.

JAXP provides basic serialization functionality which is included in the javax.xml.transform package. There are a several steps in order to perform XML serialization using JAXP as depicted in Figure 2-12. These steps comprise:

1. Creating a new instance of the TransformerFactory class.
2. Creating a transformer by invoking the newTransformer() method on the TransformerFactory instance.
3. Constructing a source object from a DOM Document object, SAX reader, or an input stream.
4. Constructing a new StreamResult object to write the resulting stream.
5. Passing the source and result objects to the `transform()` method created in step 2 [26].

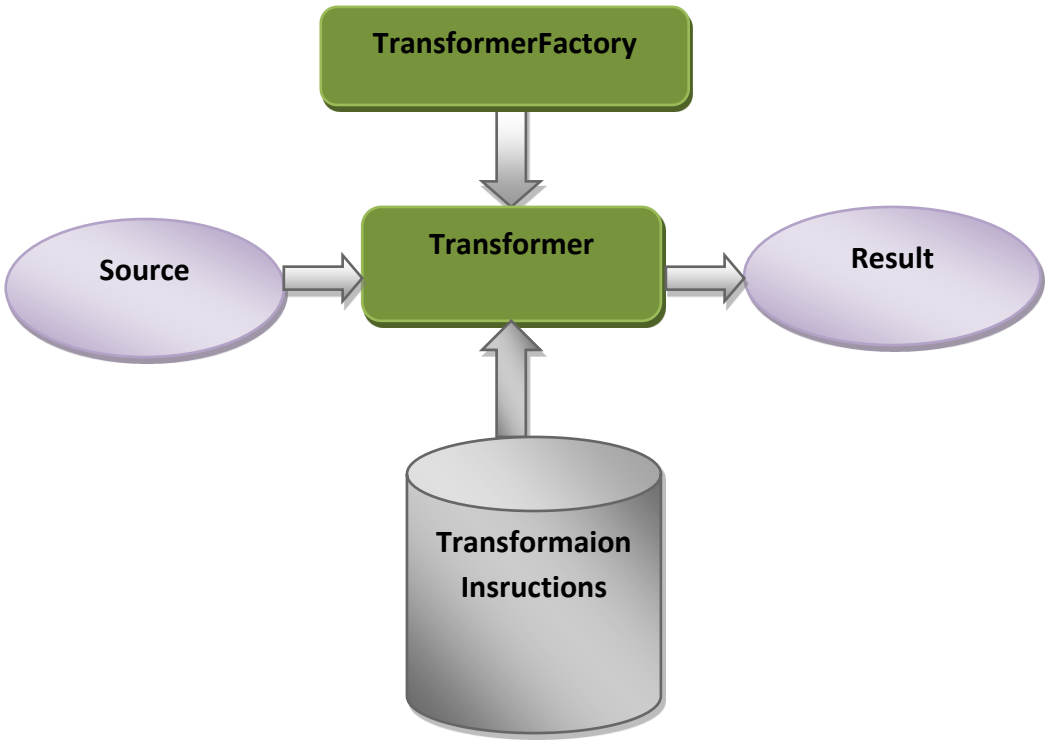


Figure 2-12: JAXP transformer [28].

2.2.4 JDOM

JDOM has been developed as an alternative to DOM. It is a Java-specific API to work with an XML object model as opposed to DOM (the latter is language-independent). Similar to DOM, JDOM provides a tree representation of the XML document and requires the whole document to be loaded into memory [26].

Unlike DOM which uses interfaces for representing different nodes of document, JDOM takes advantage of concrete classes to represent the tree nodes. This is possible because JDOM is a Java-native API developed specifically for XML processing [26]. As JDOM is Java-specific, it might be preferred by Java developers [29]. JDOM performs Java object to XML serialization. However, it is not included in the Java standard which could be a drawback as it means that there is a possibility that this API will be discontinued in the future.

Figure 2-13 illustrates the object to XML conversion performed by JDOM. First, the JDOM document is created from a SAX or DOM builder. The former is constructed from a set of SAX events, while the latter is created from a DOM tree. The JDOM document can also be directly constructed. The next step is to instantiate an XMLOutputter object and invoking

the output method of this object passing it two arguments: the first argument is the document which is to be serialized (i.e., the JDOM document) and the second argument is the stream to which the output XML is written [29].

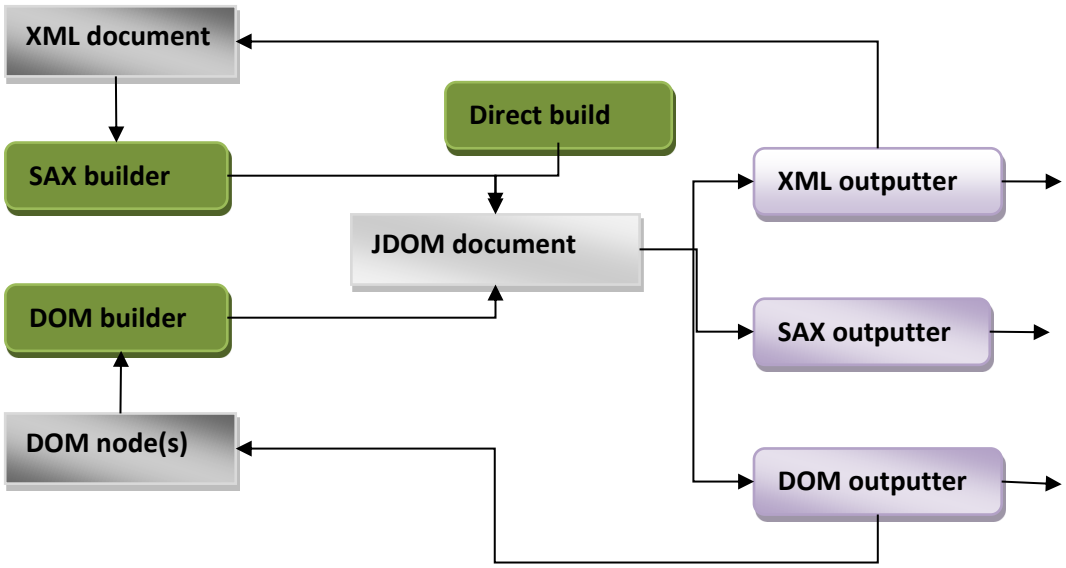


Figure 2-13: Input and output loops in JDOM [29].

2.2.5 XStream

XStream is a library to serialize/deserialize objects to/from XML. XStream offers a variety of features that facilitate object to XML conversion [30].The most important of these characteristics are:

- It is not mandatory to define getter and setter methods for the objects to be serialized.
- XStream can serialize instances of classes without any default constructors.
- XStream is not restricted to public class fields, hence private and final field serialization is also supported.
- XStream supports not only XML, but also JSON as an output format.
- Mapping which relates names in Java to their corresponding representations in XML is completely optional in XStream.
- XStream consists of a set of registered converters. Each converter has the responsibility to convert a common Java type into the XML.
- XStream serialization preserves the property order of the class instance, which means class fields are written to XML or JSON in the same order as they are defined [30].

The serialization process seems quite straightforward. One simply instantiates the XStream class and invokes the `toXML()` method by providing the object to be serialized as a parameter. However, if a field has an arbitrary or custom type for which XStream does not provide any converter, then a custom converter can be created and registered with Xstream [30].

If the programmer creates a custom converter and the class is updated with a new property, then the converter should also be updated. While this may be suitable when we have a small well-known set of classes with restricted or no changes, this approach is definitely *not* appropriate for frequently changing classes.

2.2.6 JiBX

JiBX is a data binding framework which is typically used as a persistence layer API. JiBX relies heavily on binding definitions for mapping between Java objects and an XML document, thus it may not be easy to use [31].

JiBX provides great flexibility as it allows the programmer to refactor the code over time without needing to change the XML structure each time. This means the structure of the Java objects are not tied to the structure of the XML document; hence it is possible to restructure the object classes without needing to modify the XML format used for external data transmission [31].

2.2.7 Castor

Similar to JiBX, Castor is also a data binding framework. Castor provides three different modes to marshal/unmarshal objects to/from XML: introspection mode, mapping mode, and descriptor mode [32].

In mapping mode, the programmer provides a mapping file to control the format of the XML which is to be constructed by marshaling. The mapping file, as the name suggests, defines how objects are mapped to XML [33]. This mapping file needs to contain a mapping for every different type of object which is to be serialized; hence the programmer might perceive it as not easy-to-use.

2.3 Related work

Currently, there is no algorithm in the public domain for generic Java object serialization to the FXML representation. However, as has already been noted, FXML is an XML-based markup language. Hence, in this section we concentrate on earlier XML serialization research.

A research study was conducted regarding the application of the Java bean components to implement visualization and 3D modeling in Java, see [34]0. The primary purpose of this research was to use Java bean components to enhance the functionality of the available 3D modeling languages. Virtual reality modeling language (VRML) is the language of choice in the aforementioned research. Similar to an FXML file whose hierarchical structure parallels the hierarchical structure of a scene graph, a 3D model in VRML is also defined by a scene graph. Using this approach a programmer implements each node object of the 3D-model as a Java bean component, thus each node object's field corresponds to a Java bean having the same name and property value as the node object. The Java bean property descriptor of the Introspection API has been used to describe the properties. A mechanism was also implemented to extract the changed properties of the VRML-file components. They constitute properties for which the Java bean components should be implemented.

Another research study deals with algorithm analysis and design regarding subjective on-line tests which is presented in [35]. The algorithm proposed to handle the mentioned problem was to investigate in parallel the XML document that is related to a standard answer (which has already been provided) with the XML document dealing with a student's answer. During the next step, a tree model is generated for each XML document in order to make it possible to access it programmatically. The proposed tree model uses the DOM model to traverse two XML files in parallel using depth-first traversal (DFS). During the traversal, each node from the standard-answer DOM tree is compared with the corresponding node in the student-answer DOM tree. Grading is based on the result of this comparison.

To the author's knowledge, this thesis is the first work considering Java object/bean serialization to FXML representation. However, the work discussed above was used as a reference for the solution proposed in this work.

3 Methodology

This chapter describes the research approach taken to address the problem presented in Section 1.2 on page 2. The framework that encompasses the whole processes required to conduct this research is introduced and the method that has been chosen during each step in this process is discussed and argued. A qualitative research approach was adopted in order to achieve the research goal or sub-goals necessary to addressing the research problem.

3.1 Research approach

The steps undertaken during this research to fulfill the research objective can be categorized into two phases. The first phase was a literature study, while the second phase utilized the design science research method. Both of these steps will be described in more detail in the following two subsections.

3.1.1 Literature review

This step includes investigating related work as well as gaining the required background information that was necessary to conduct this thesis project. The literature study helped to determine the relevance and originality of the research. As there was no existing algorithm in the public domain to provide generic object/bean serialization to the FXML, the related work section (see Section 2.3) identified some studies pertaining to XML serialization mechanisms.

In particular, the literature study provided a summary of the current state of the art regarding the specific research problem that revolves around the JavaFX FXML markup language and an enumeration of the FXML specification. Furthermore, different APIs for serialization of objects to XML were identified and considered. The goal of the literature study was to identify relevant serialization APIs that might ease implementation of generic FXML serialization. There exist many APIs and third party libraries that deal with object to XML serialization. Hence, to fulfill the aforementioned goal, we consider a number of metrics against which the available APIs are evaluated. In this way a set of APIs or third party libraries that are widely utilized in the domain of the Java object to XML serialization and which fulfill the evaluation criteria, are chosen as potential candidates for further investigation.

Literature review and Internet searching were the methods of choice for conducting this part of the project since one of the most reliable means to gain knowledge about a particular API is the API's documentation. In order to analyze different APIs based on the evaluation metrics, a theoretical analysis was performed. The API's specification and features as well as samples provided by the APIs' website, efficiently provided input for this research. This information enabled the potential candidates to be selected and categorized.

A few experiments were done to gain insight into how some of these APIs perform when it comes to object serialization. However, the experimental approach was not used as the primary method in this phase to select the candidate serialization APIs since the theoretical analysis was sufficient to evalaute the APIs according to the specified metrics and to decide which specific APIs to choose.

3.1.2 Design science research method

From the theoretical analysis conducted in the previous phase, the most promising serialization alternative was chosen from among the previously identified potential candidates. The selection process was based on the evaluation metrics that will be discussed in Section

3.1.3. The selected alternative contributed to the design decisions for the solution developed in this thesis project.

The second phase of this research project was based on a qualitative research method. A design science research approach was used in this phase to achieve the remaining research sub-goals, specifically the design and implementation of an algorithm that facilitates Java bean to FXML serialization, as well as verification and evaluation of the results.

Design science research is research method in which the contribution to the existing knowledge and addressing the research problem is conducted by means of constructing an innovative artifact [36]. An information technology artifact could be models, algorithms, demonstrators, design guidelines, etc [37]. The artifact which is to be constructed during this research is an algorithm which is capable of serializing the Java object/bean into FXML.

Figure 3-1 demonstrates the framework which has been used during this research in order to designing the desired artifact. The returning arrows, so-called circumscription, imply iterative processes within this framework.

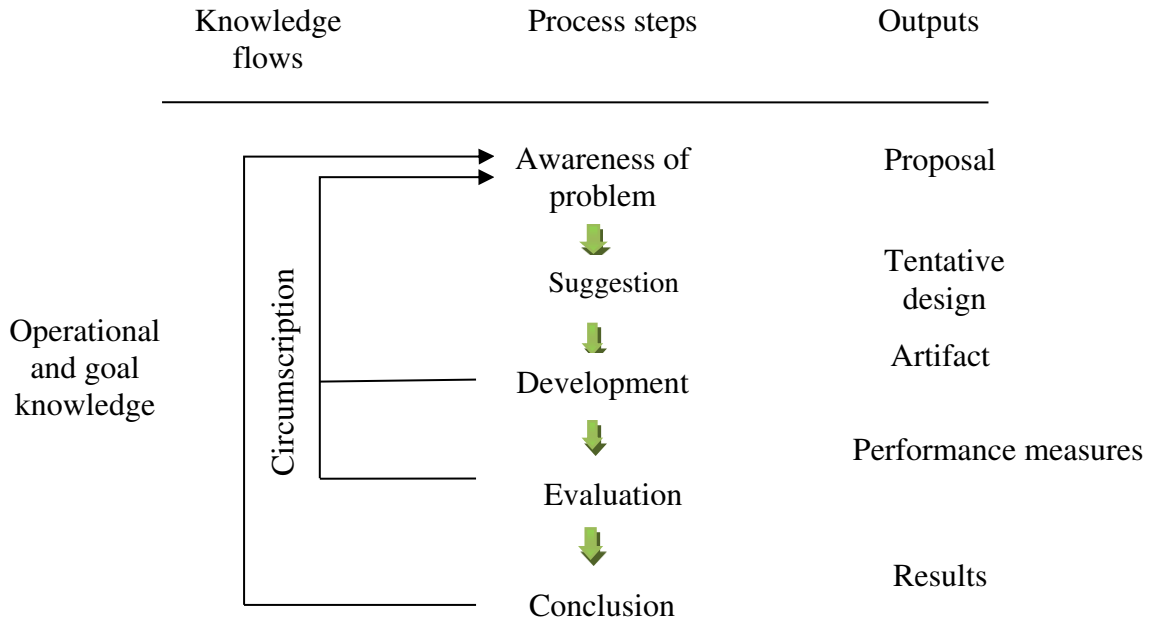


Figure 3-1: The design science research framework [36].

3.1.2.1 Awareness of problem

During the first step called *Awareness of problem*, two activities are performed: first, the problem is defined and secondly the solution objectives are specified. These objectives constitute the set of requirements whose fulfillment is the indication of successfully completed research.

In order to define the problem, the natural method of choice was to review Ericsson Research's internal documents for the visualization platform and to examine the way an application is defined in this visualization platform. The qualitative data gathered during this phase provided a basis for defining the project's research objectives.

It should also be mentioned that although the problem is defined within the context of a particular application platform, the solution which is to be provided, should *not* be restricted to this platform. Instead, the solution should be a generic one; hence it should also be applicable to other visualization platforms to facilitate JavaFX object serialization to FXML.

The second activity needed to get sufficient awareness of the problem was to define the objectives for the solution and to define the requirements. The qualitative data collected from the problem definition phase was analyzed by document analysis and as mentioned previously, an extensive literature study was conducted to assess the state of the art in the research domain.

The result of the literature review and document analysis revealed that in this research area, there were two kinds of requirements that need to be considered during the design process: first, those requirements that are extracted from the business needs (as these requirements concern the desirable functionality that the artifact should support) and second, the requirements that are related to the construction of the artifact itself. The later set of requirements was derived from the results of the literature review – enbling the project to build upon the current state of the art.

3.1.2.2 Suggestion

The next step in this design science research was to propose a tentative design of the algorithm based on the identified problem and specified requirements. This algorithm should fulfill both the business needs and the construction requirements.

3.1.2.3 Development

The designed algorithm is developed in this step. In fact, the processes of design and implementation (development) are performed iteratively so that the designed algorithm would be refined incrementally according to the required functionality and the design requirements. In other words, a test-driven development approach is used. This will be further described in in Section 3.1.2.4. To make the iterative approach more apparent, Figure 3-2 is illustrated which has the same nature as the previously mentioned design science research framework depicted in Figure 3-1 in terms of the required processes to conduct this research.

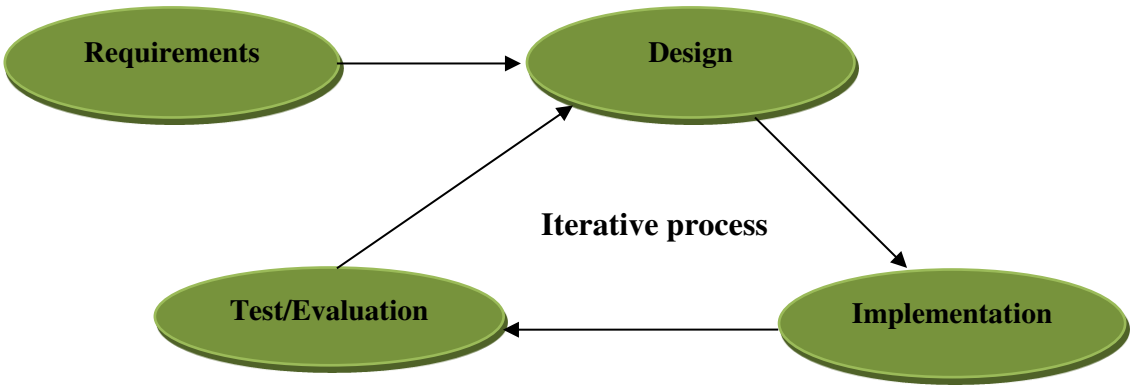


Figure 3-2: The research method.

3.1.2.4 Evaluation

The next step during this research is to evaluate the developed artifact against the specified requirements to ensure that all the requirements are fulfilled. There are different methods to evaluate a developed artifact, e.g. interview, questionnaire, testing, etc. [37]. Among these, black-box testing and a questionnaire were chosen. Each of these will be described below.

Black-box testing is one of the evaluation methods of choice since it guarantees that the software complies with the stated requirements under the assumption that the tests are carefully designed to verify these requirements.

The JUnit testing framework was used during our test-driven development [38]. A test case is provided that contains different test methods. Each test method aims to evaluate a particular requirement.

A utility is also provided that does the actual verification based on XMLUnit testing [39]. According to Bacon and Bodewig, XMLUnit "enables JUnit-style assertions to be made about the content and structure of XML" [39].

The overall evaluation method utilizes a set of FXML files, where each file is designed to verify that a particular requirement is provided by the serialization algorithm. FXML files that constitute the expected results of the serialization are loaded and verified using JUnit and XMLUnit testing. Note that during the evaluation of these requirements, the design requirements are also implicitly evaluated (since they constitute the basis for the proposed solution). Unless the requirements are met a completely different (i.e., not even similar) FXML document would be generated that when compared with the original one leads to a test failure.

The set of functionality specifically provided by XMLUnit is:

- XMLUnit verifies not only identical FXML files but also similar files, thus the serialization result and the expected result do not need to match exactly. This functionality of XMLUnit is helpful since the order of elements in the serialized FXML and the expected FXML might differ. This happens particularly for the object/bean properties (corresponding to the FXML property elements) that are introspected by the bean property introspection. The bean property introspector reads the properties of an object/bean in the alphabetical order. Potentially, this could be a problem for the verification of the generated FXML. However, XMLUnit does not care about the order of the elements (property elements) which eliminates this as an issue.
- Another useful property of XMLUnit is that, it also does not care about the order of the import processing instructions. Processing instructions may appear in different orders in the serialized FXML file and the original FXML. As long as the algorithm provides the serialized FXML with all the required import processing instructions that are included in the expected FXML, then their order does not matter.
- XMLUnit makes it possible to ignore white space in original/expected FXML document for the purpose of verification.
- XMLUnit also makes it possible to ignore comments in the original FXML file.
- XMLUnit instantiates a Diff object in order to compare the expected result of the serialization with the actual result. It is also possible to get a detailed description of two FXML files differences using the DetailedDiff class [39].

In order to evaluate the conformation of the designed algorithm with the stated requirements from a functional perspective, test coverage was also performed on the algorithm together with unit testing. Together they indicate the degree to which the algorithm conforms to the designed test cases. The Java code coverage tool used for this purpose is EclEmma which is a free code coverage tool for Eclipse [40].

The statistics obtained using EclEmma indicates the statement coverage of the serialization algorithm including covered instructions and the dead code (referred to as missed instructions).

The third approach to evaluate the compliance of the designed serialization function with the requirements is that the resulting code should be reviewed or tested by a small group of developers that work with the visualization platform on a daily basis. This group of people was asked to answer a quationnaire covering their experience when working with the FXML serialization function, the errors or inconveniences that they might have encountered during working with it, and any obvious disadvantages that they noticed with the FXMl serializer. The questionnaire used for this evaluation is included as Appendix A.

3.1.2.5 Conclusion

During this step, the results of the evaluation are analyzed. If the conclusion is that the requirements are *not* met based on the outcome of the defined evaluation metrics, then the iterative design process is repeated until the results of the evaluation reveal that the specified requirements are fulfilled.

3.1.3 Evaluation metrics

As mentioned in Section 3.1.1, a set of metrics has been utilized when evaluating candidate APIs or libraries to be used during development of the algorithm. These criteria are listed in priority order as follows:

- A third party library or API is preferred over another alternative if it has a relaxed license, i.e. GNU Lesser General Public License (LGPL), BSD, or similar license. Commercial libraries (libraries that are associated with a license cost) or GNU General Public License (GPL) libraries are unsuitable for this project.
- A third party library or API is preferable over another alterative if it reduces the amount of code required to design the FXML serialization algorithm due to decreasing the amount of implementation work and hence reduces the need for future source code maintenance.
- A third party library or API is preferred over another alternative if the library or API is a software project that has not been discontinued and has a clear roadmap for the future and an active developer community.
- A third party library or API is preferable if the library or API is well documented and hence is possible to use or understand based on the documentation provided.
- A third party library or API is preferred over another one if the library or API is native Java and not a pre-compiled dynamically linked library with Java bindings.

3.2 Environmental resources

Java and the Java Development Kit (JDK) 7 will be used as the implementation language. The integrated development environment (IDE) used for development is Eclipse 4.2.2 SDK which includes both e(fx)clipse 0.8 as well as Subclipse 1.8 that are used during this project. The e(fx)clipse Eclipse plug-in is used to support JavaFX tooling and Subclipse supports Subversion (for version control) within the Eclipse IDE.

4 Requirements

According to the research method which was introduced and discussed in the previous chapter, defining the requirements is the next step after providing the required foundation and background for the research. There are two main resources from which the collection of requirements needed during this project is derived. Consequently, the requirements are categorized into two groups: design requirements and functional requirements.

4.1 Design requirements

Design requirements refer to those requirements that are considered to construct the algorithm. They were extracted from the knowledge obtained from the background information and related work concerning: (1) XML serialization and related APIs and (2) the JavaFX and FXML specification in particular.

4.1.1 Traversing the scence graph

To serialize a JavaFX scene graph into its correspondent FXML, a preliminary requirement is to visit every scene graph object (usually referred to as *node)*. In other words, it is necessary to traverse the whole scene graph, node by node, while avoiding loops. Graph traversal is required to construct an FXML file in which every element corresponds to a visited scene graph object and the attribute of that element corresponds to the property of that object in the scene graph.

4.1.2 Detecting changes in the properties of the scene graph object

One of the algorithm's design requirements is to determine the properties for which a particular object or node has a value other than the default value of that property. The reason is that the result would not be changed if a property is set to a value equal to the initial or default value of the property. Such an operation leads to a verbose FXML and adds no value.

4.1.3 Generating valid FXML from the travered scene graph

In order to create a valid FXML document and to define a logical structure for it, it is necessary to map the scene graph nodes to FXML elements as well as to map bean properties to the FXML attributes.

4.2 Functional requirements

The functional requirements on the algorithm are specific to the desired functionality. They comprise the FXML-specific requirements that should be supported by the serialization algorithm. These requirements are specified based on the business needs; hence they all should be supported in order to make the serialization algorithm work with the generic visualization platform. These requirements are:

- Handle the import statements in the FXML files and generating them from the classes used;
- Support the primitive or atomic properties and serializing them to FXML property elements;
- Support lists, particularly the ObservableList of FXCollections collections;
- Support immutable objects (i.e., objects that use builders for creation);

- Support static properties;
- Support fx:controller; and
- Create unit test cases that can be used to verify the functionality of all the previously mentioned serialization features and can also be used as regression tests.

Furthermore, there are some other FXML features that need to be supported in order to construct an algorithm that provides complete FXML serialization functionality. However, they are out of scope of this thesis since they are not currently supported by the WYS/WYG run-time GUI editor which was constructed in a parallel thesis project in order to provide a built-in layout designer [41]. Hence, they will be mentioned in section 7.4 as potential future work.

5 Design

This chapter begins by describing the design of a proposed solution for the serialization problem. A subset of the APIs introduced in the second chapter is selected based on the results of a theoretical analysis according to the evaluation metrics presented in Chapter 3.

This chapter continues with the design of the proposed solution considering each required feature (as enumerated in the functional requirements) that needs to be supported by the designed solution. Consequently, the chapter is divided into two sections: design solution and functionality-specific solution.

5.1 Design solution

Figure 5-1 illustrates the proposed algorithmic solution. First, the solution is described generally and then the design is elaborated in order to introduce those APIs that have been selected to handle the different parts of the proposed solution.

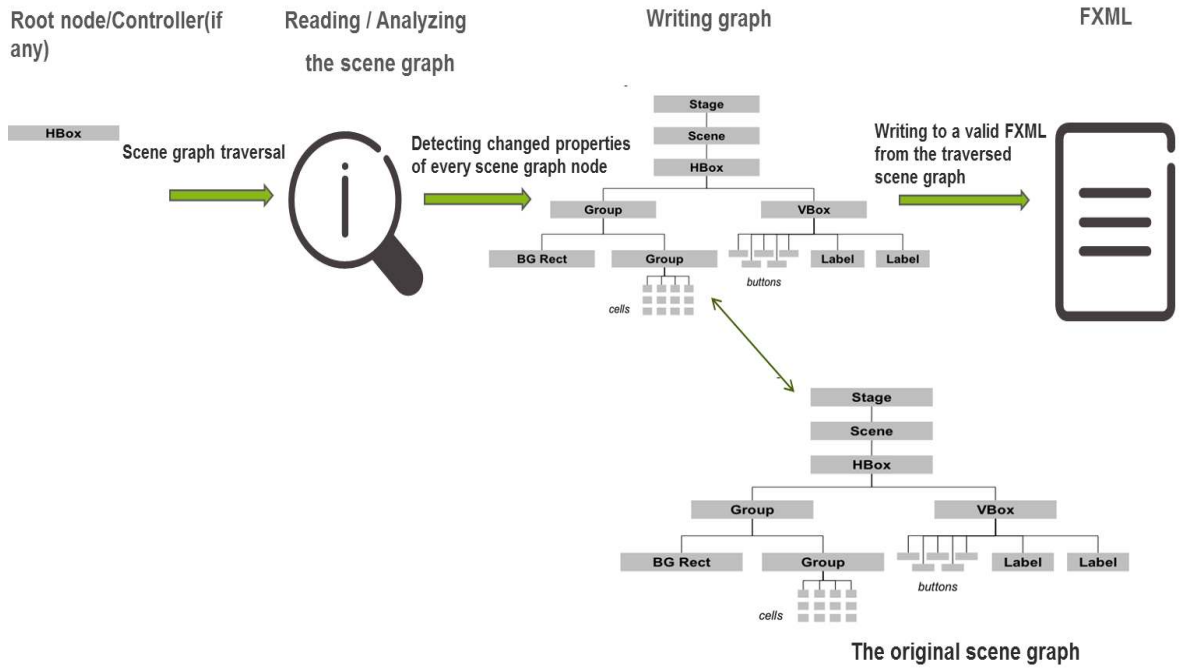


Figure 5-1: The proposed solution.

Figure 5-1 depicts the steps to be undertaken in the proposed algorithm. First an FXML file is loaded by the JavaFX FXMLLoader class. The root node of the JavaFX scene graph is then obtained from the loader instance. The root node constitutes the entire scene graph which corresponds to the loaded FXML document.

In the proposed solution, the serialization function takes this root node as its input argument. It should also be noted that there might be an optional controller class which is used as an event handler for the user interface elements embedded within the FXML document [13]. In this case, the controller class should also be passed as the input argument to the serialization function.

Starting from the scene graph's root node, the complete scene graph is read and analyzed node by node. The purpose of reading and evaluating each node is to decide what to write into

the FXML document. In order to ensure that every node of the scene graph object is visited, the algorithm needs to traverse the entire scene graph.

Finally, the resulting FXML document is compared to the FMXL file which was initially loaded to generate the scene graph. The result of this comparison determines whether the two FXML documents are similar (but not necessarily identical) or not. Similar FXML documents should have equivalent scene graphs.

After each design requirement is discussed and presented, the achievement of each requirement is examined with respect to solving one part of the problem using the proposed solution shown in Figure 5-1.

5.1.1 Traversing the scene graph

In order to traverse the scene graph and process each node during the traversal, it is necessary to apply a traversal algorithm. Two alternatives have been investigated: depth-first search (DFS) and breadth-first search (BFS) [42, 43].

The DFS algorithm, as the name implies, traverses the graph as far as possible along each branch before backtracking. Starting from the root node, it traverses each branch of the graph following the edges. Each edge links one node to its successor node (referred to as a child node). The algorithm backtracks whenever either a node has no children (usually called vertex) or it has children and all of them have already been visited. In this case, the search is continued from the nearest ancestor that has unvisited children. This process continues until every scene graph node has been visited [42]. Figure 5-2 shows the depth-first iteration order.

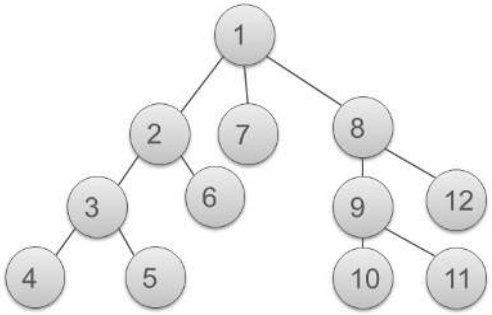


Figure 5-2: Depth-first iteration order.

BFS on the other hand, as illustrated in Figure 5-3, takes a different approach. Beginning with the root node the traversal algorithm visits every node which is directly reachable from the root node, i.e., all nodes at the same distance from the root node. For every visited node in turn, all of its neighbors are visited and this process continues until all the nodes have been visited [43].

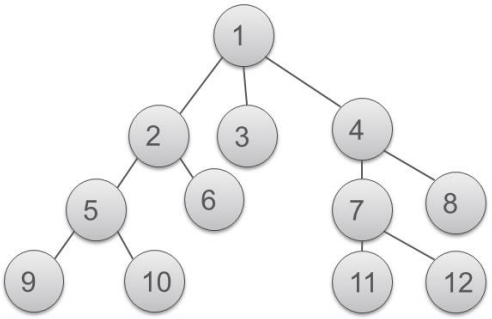


Figure 5-3: Breadth-first iteration order.

DFS was chosen for traversing the scene graph based on the type of problem which we are dealing with. One argument for selecting DFS is that DFS starts by going to each leaf of the graph which makes it a good alternative for traversing our scene graphs. There exist two implementation approaches for DFS: recursive and iterative DFS. One drawback with a recursive implementation is that it can potentially fail due to stack explosion when the number of scene graph nodes is extremely high. For this reason, the iterative DFS was chosen for the implementation.

5.1.1.1 Design considerations during the depth-first traversal

Iterative DFS is implemented using a stack data structure. Taking the root node of the scene graph, a depth-first traversal starts by pushing the visited root node to the stack. Then, one of the issues during the traversal is to identify all the transitions that lead from one node to its children in the graph. For this reason it is desirable to expose the internals of every node in the scene graph. By node internals we refer to the fields, methods, constructors, etc. of the node object by which we can deteremine the transitions that lead to the children of this node.

Java SE provides two APIs to examine the internal structure of Java objects (or beans) that have already been investigated and discussed in Chapter 2 with regard to their features and functionality: Java Reflection API and Java Introspection API.

These APIs are part of the Java Standard package and completely fulfill our needs to extract the required information from one object for the sake of identifiying its children in the graph. Therefore, these APIs are considered as candidates to deal with reading and analyzing the scene graph.

There are two basic sorts of nodes in the scene graph: *leaf nodes* and *branch nodes*. Branch nodes refer to nodes that have children, while leaf nodes do not. Figure 5-4 illustrates these two scene graph node types.

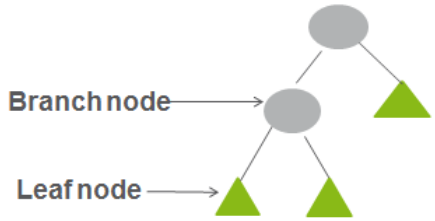


Figure 5-4: Scene graph node types.

The algorithm needs to detect leaf nodes in order to initiate backtracking. Leaf node detection is done based on inspection of the node's type. The scene graph leaf nodes can be categorized into three groups:

Wrapper classes A node which is of a primitive wrapper type represents a leaf node. These classes are used to wrap *eight primitive Java types* plus *void* e.g. *Integer* is the wrapper class for the *int* primitive type.

Enum A node, whose class is an extension of the java.lang.Enum class, is counted as a leaf node.

String A node which is a String instance does not have any children in the scene graph hence it is considered as a leaf node.

Two lists need to be populated during the graph traversal:

- One list is dedicated to storing the results of using Reflection/Introspection APIs to learn the children of each scene graph node.
- One list is utilized in order to store all the visited nodes of the scene graph, thus making it possible to keep track of the visited children in order to avoid loops.

The DFS algorithm backtracks when:

- It visits a leaf node or when
- It reaches a node which is not a leaf node, but has no children that have not already been visted.

It is good to mention that in the non-recursive DFS, backtracking is carried-out when a node is popped from the stack. In this case, the nearest ancestor of the node would be automatically on top of the stack and this will be the node from which the DFS algorithm resumes.

One of the primary considerations during the algorithm design and implementation is that it is necessary to uniquely identify every node of the graph. To achieve this goal, when a node is visited, it is wrapped within a container that contains additional information. The combination of additions to the node object makes the node unique. The container object is composed of three fields in addition to the node which is being visited (which is considered to be the *current node*). These three fields and the current node of the container are:

1. The nearest ancestor (referred to as the *parent node*) of the current node.
2. The current node object.
3. A string representing a property of the parent node whose (`getter` *method*) invocation on the parent node results in the current node object. It should be highlighted that the property is obtained using the PropertyDescriptor of the Introspection API and the `getter` method of the property is invoked on the current node object using reflection.
4. The parent node's container/wrapper: In order to provide a link between transitions or edges of the graph, information from the node's parent should be added to the node's container. This information is provided by the container of the node's parent. By taking advantage of this nested node wrapper/container, the unique path from the root node to the current node is provided.

Figure 5-5 shows the structure of each scene graph node which is composed of the four fields that have just been described. The numbers in the figure are related to the corresponding field.

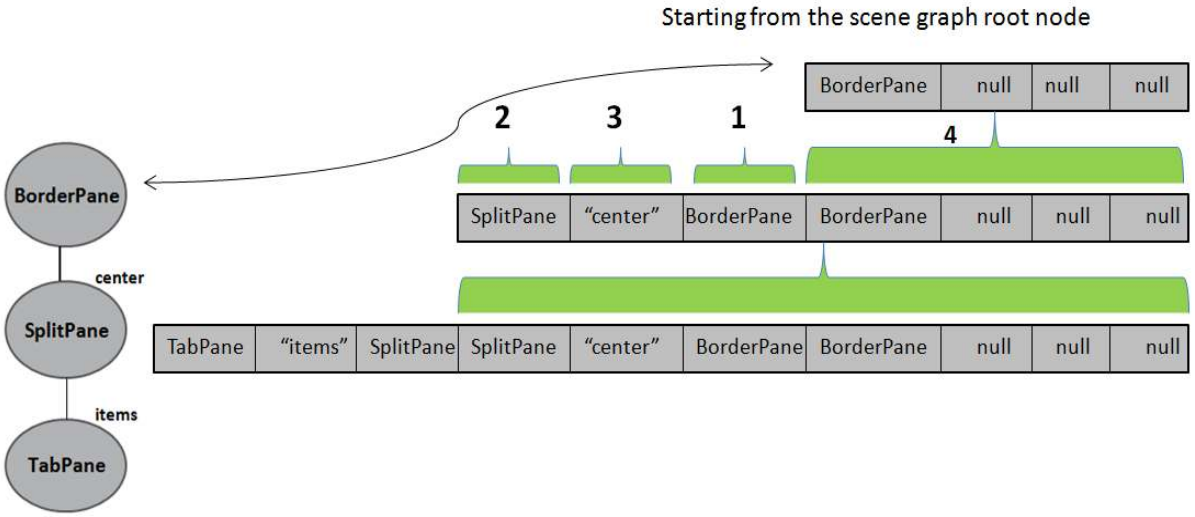


Figure 5-5: The structure of each scene graph node.

5.1.2 Detecting changed properties of the scene graph object

As has already been noted in the design requirements, it is necessary to detect when the properties of an object/bean have a value different from the default value of that property. The typical approach with Java beans is to create a new instance of the object and compare the newly constructed object with the original object property by property. To obtain the value of each property, the property's `getter` method is invoked on the object using reflection. The properties of the new object instance all have the default values. Among all the properties of a Java object/bean only those with non-default values should be written to the resulting FXML document.

Simply creating a new instance of an object is feasible for classes that have a default constructor. However, not all Java classes have a default constructor. There may be objects that are constructed using factory methods or those that require builders for construction. Factory methods and builders will be discussed in more detail in Section 5.2.

The non-default value of an object/bean property in the scene graph which is obtained using a bean property descriptor and introspection represents:

1. A leaf node,
2. A collection of nodes, or
3. A standard node (a node which is neither a leaf node nor a collection of nodes).

5.1.3 Generating valid FXML from the traversed scene graph

In order to generate vail FMXL during the traversal of a scene graph we have to consider the API to be used and the specifics of our design. Each of these will be discussed in the the following paragraphs.

5.1.3.1 API selection

To deal with writing the serialized scene graph, various frameworks and APIs have been introduced in the second chapter of this thesis. These frameworks and APIs are specifically concerned with object/bean serialization/deserialization to/from XML.

It is essential to point out that for this particular design, we decided to write the FXML *during* the traversal which means whenever a node is read/visited, it needs to be analyzed and a decision must be made regarding whether to write a serialized verison of it into the FXML document or not.

Table 5-1 illustrates the evaluation results of the APIs presented earlier. These evaluation metrics have been used in order to select a candidate to handle the generation of FXML.

Table 5-1: Object serialization/deserialization to/from XML using different APIs.

	Relaxed license	Code reduction	Requires any mapping, binding or converting	Continuous project and clear roadmap for future	Well documented	Native Java
XStream	open source software, available under a BSD license	No	Yes	Last release: January 19, 2013 XStream 1.4.4 55Commits during the last 12 months	Yes	Yes
JiBX	This license (BSD form) covers the actual JiBX code itself. Also includes BCEL license and XPP3 license	No	Yes	Last commit: almost 2 years ago	Yes	Yes
JAXP **(Included in JavaSE)**	CDDL(Common Development and Distribution License)	Yes	No	Last commit over 2 years ago	Yes	Yes
JDOM	BSD license	Yes	No	Most recent commit 4month ago	Yes	Yes
Castor	Apache-2.0 and BSD-4-Clause-UC	No	Yes	Most recent commit about 1 month ago	Yes	Yes

When it comes to the license considerations, XStream, JiBX, JDOM, and Castor APIs have a BSD license, while JAXP has been developed under a Common development and distribution license (CDDL). All of these fall into the relaxed license category, hence they are all acceptable.

Unfortuantely, in terms of code reduction XStream, JiBX, and Castor seem unsuitable alternatives since:

- XStream requires the creation and registration of new converters for custom types. As mentioned previously, converters are responsible to associate Java types with XML elements. Therefore, XStream might not contribute to a code reduction since it requires a converter definition for each custom class.

- Using JiBX, it is necessary to provide binding definitions in order to map Java objects into XML. When using this framework for the XML serialization API the expected amount of required code would increase with the number of different Java classes.
- Castor also requires a mapping file to describe how objects are to be mapped to XML. Therefore, the code reduction criteria is not satisfied when using the Castor API.

DOM and JDOM APIs on the other hand, do not need any extra mapping, binding, or converting in order to generate a clean FXML document from the Java objects/beans. Accordingly, the code required to construct an FXML document from the scene graph would be significantly less using DOM or JDOM than XStream, JiBX, or Castor.

It should be noticed that, all the above APIs require an implementation effort to produce XML from a Java object. However, XStream, JiBX, and Castor also need converting, binding, and mapping respectively in order to output an FXML document. Hence, the code reduction requirement favors producing the FXML from Java objects/beans with a minimum implementation effort.

From a continuity point of view, all APIs more or less have a clear future roadmap. Although it seems like XStream, JDOM, and Castor developers have been recently more active compared with the DOM and JiBXdevelopers based on the statistics showing the most recent commits to these libraries. However, the lack of activity may reflect the maturity of the library.

Finally, all five APIs are well documented and are native Java libraries.

In conclusion, JAXP and JDOM are the most promising candidates based on the defined evaluation metrics. They both solve the serialization problem and are equivalent in terms of the required implementation effort. However, JAXP was chosen since it is already built-into Java. Hence, no external JAR library is needed and it is unlikely to be discontinued. JDOM is not included in Java SE and might not be continued by developers in the future.

According to the description provided for the JAXP API in chapter 2, DOM was selected to be used with JAXP since it makes it possible to write into a FXML document based upon a tree model of the document and then using JAXP transformer's transform method to translate the tree model (DOM) into a textual representation (FXML).

It is worth mentioning that the Java architecture for XML binding (JAXB) is another popular framework in which Java classes are mapped to XML using JAXB annotations and this could be an alternative for serialization [44]. However, JavaFX is distributed as a pre-compiled JAR file and hence cannot be changed or annotated, thus JAXB is not an alternative to solve the object to XML serialization problem and it has not been considered as an alternative in Table 5-1.

Figure 5-6 summarizes the complete design solution.

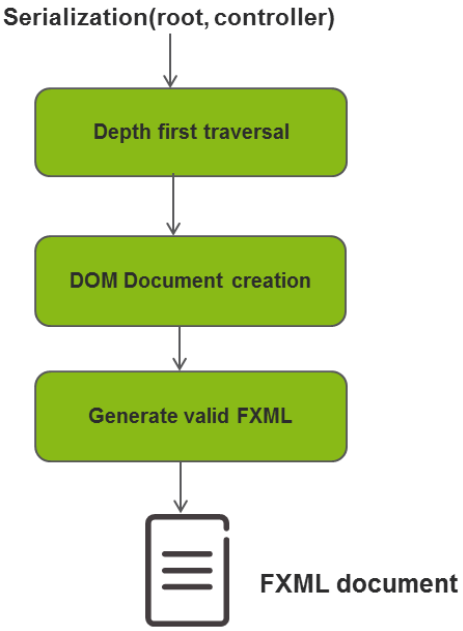


Figure 5-6: The design solution flowchart.

5.1.3.2 Design considerations for writing the FXML document

In order to generate a tree model (DOM) from the visited scene graph nodes that maps perfectly to FXML, the algorithm is designed so that a grandparent-parent-child relationship is defined as illustrated in Figure 5-7 in which:

- A child represents an FXML element corresponding to the node which is currently visited in the scene graph using DFS.
- A parent represents a property element in the FXML document corresponding to the property in the scene graph which is invoked (using reflection) on the current node's parent and leads to the current node. It should be emphasized that the algorithm writes the object/bean properties in the form of property elements and not property attributes in the FXML document.
- A grandparent is an FXML element corresponding to the current node's parent in the graph.

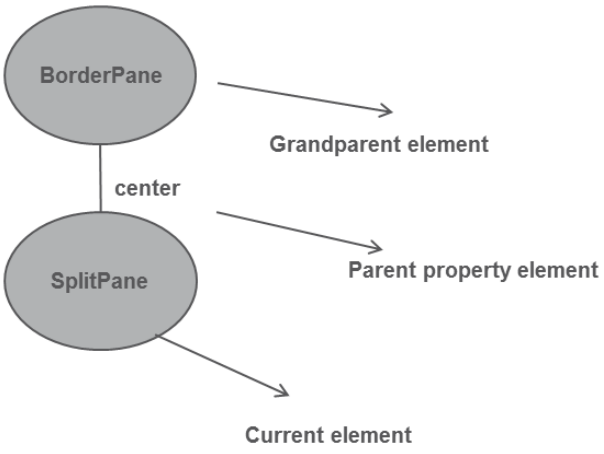


Figure 5-7: Mapping the scene graph nodes to the DOM elements.

After mapping one graph node into a DOM element, the algorithm transforms the DOM model into a valid FXML document. One design consideration is that the FXML document is divided into two parts: an *import document* that involves import processing instructions (i.e. fully-qualified class names of the utilized classes in the body of the FXML document) and the *node document* or the body of FXML document which is composed of different elements and attributes (also handled by property elements).

The demarcation in Figure 5-8 depicts the generated FXML for the partial scene graph shown in Figure 5-7 which was composed of two nodes (i.e. BorderPane and SplitPane) in which the BorderPane instance is the scene graph root node. If the SplitPane object is the currently visited node in the scene graph, then the SplitPane element would be the correspondingt FXML element, *center and* BorderPane elements refer to its parent and grandparent elements respectively. The demarcation in the import document involves fully-qualified class names for the Borderpane and SplitPane instances included in the node document.

Figure 5-8: FXML document structure.

One of the primary concerns regarding writing to the FXML document is to find the correct grandparent element in the FXML to which the current node's element should be added. For instance, if there are multiple BorderPane elements in the FXML document, then how can we detect the correct one to which the SplitPane element should be added? The algorithm solves this issue by mapping every object to a DOM element during the traversal and then retrieving the unique grandparent element during the FXML generation. The retrieved grandparent element is associated with the currently visited node's parent which was already mapped during the traversal.

The other design consideration during the FXML construction is that for every element which is to be added to the FXML document, first the algorithm should check if the parent element has already been written to the FXML document. If this is the case, then the algorithm should simply add the current element to the FXML document (without writing the parent element). It is necessary to check for this since a node might have multiple children in the scene graph; hence an element corresponding to every child should be added to the *same* parent element in the FXML document, otherwise the test fails.

5.2 Functionality-specific solution

Each of the subsections of this section describes a specific solution to provide a required functionality.

5.2.1 Handling the import statements in the FXML document

As has already been noted, the serialized FXML document is composed of two documents: an *import document* and *node document*. Import statements (referred to as *processing instructions*) should be added to the import document.

Import statements are required in order to resolve short class identifiers in the FXML hierarchy, but can be omitted if fully qualified class names are always used in the body of the FXML document. However, that leads to very verbose FXML and poor readability. To provide the FXML document with the appropriate import statements, the algorithm adds the fully-qualified class name of every node which is visited during the traversal to a set. Subsequently, the import document is populated with import processing instructions with each added as a separate line. Recall that every FXML element corresponds to a JavaFX class. Figure 5-9 shows six import statements used in an FXML document to support those six different classes which are utilized in the FXML document.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.BorderPane?>
<?import javafx.scene.layout.Pane?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.paint.Color?>
<?import java.lang.String?>
<?import java.lang.Double?>
```

Figure 5-9: import processing in structions.

5.2.2 Support for primitive or atomic properties

Eight primitive types (generally types corresponding to the scene graph's leaf nodes) are handled in FXML using the *String* and/or *primitive wrapper classes*. The fx:value attribute is used to hold the string representation of the primitive or atomic property's value.

During the description of earlier design steps, it was mentioned that the value of every scene graph node is stored in an object instance. Consequently, when the algorithm encounters a node of a primitive type, the JVM automatically boxes it into the corresponding wrapper type which is a reference type (e.g. a double primitive type is boxed into Double which is the primitive wrapper class). Hence, the element name in DOM for the visited node would be the primitive wrapper type (according to the FXML syntax) and the value of fx:value attribute would be the string representation of the node's primitive value. In Figure 5-10, the FXML syntax of primitives or atomic properties is illustrated.

```
<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.Double?>
<?import java.lang.String?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.BorderPane?>
<?import javafx.scene.text.Text?>

<BorderPane xmlns:fx="http://javafx.com/fxml">
    <prefWidth>
         <Double fx:value="320.0" />
     </prefWidth>
     <bottom>
         <Label>
            <text>
                <String fx:value="myLabel" />
            </text>
          </Label>
     </bottom>
     <center>
           <TextField>
               <id>
                <String fx:value="textfield" />
               </id>
               <text>
                 <String fx:value="hello" />
               </text>
            </TextField>
     </center>
     <left>
        <Label>
            <text>
               <String fx:value="who are you?" />
             </text>
         </Label>
     </left>
     <top>
        <Text>
            <text>
               <String fx:value="java-Buddy" />
            </text>
         </Text>
       </top>
</BorderPane>
```

Figure 5-10: FXML primitive or atomic properties.

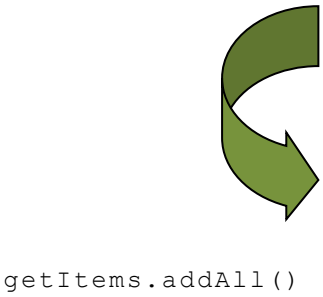
5.2.3 Support for lists, particularly the ObservableList of FXCollections collections

The ObservableList is one of the JavaFX specific collections. Scene graph analysis revealed that properties of ObservableList type could have setters and getters or only getters. Therefore, there are two categories for JavaFX collections: collections and FXCollections collections.

Using the API documentation for objects of type ObservableList as well as the scene graph, analysis showed that FXCollections instances are used to handle JavaFX collections that have both getter and setter methods. Otherwise, if there exist only getters for the properties of the ObservableList type, then the FXML parser determines that the property type is assignable from *Collection<T>* and hence there exists an `add(T)` method which can be invoked to populate the collection, hence no instantiation is required for any type other than T.

The code snippet in Figure 5-11 shows an example of a collection in FXML. In this example, invoking the `getItems` method on the Menu instance returns an ObservableList of MenuItems which is not writable. Hence, it is only possible to add an item to this

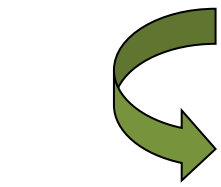
ObservableList (but not replace the list instance) and the only required instantiation is related to the instances of MenuItem type.




```
<Menu>
    <text>
        <String fx:value="File" />
    </text>
    <items>
        <MenuItem>
            <text>
                <String fx:value="New" />
            </text>
        </MenuItem>

        <MenuItem>
            <text>
                <String fx:value="Open" />
            </text>
        </MenuItem>

        <MenuItem>
            <text>
                <String fx:value="Save" />
            </text>
        </MenuItem>
    </items>

</Menu>
```

Figure 5-11: An FXML collection of MenuItem objects.

On the other hand, classes that have a setter method that makes it possible to replace the ObservableList instance should be written to the FXML document using the FXCollections element. This element has an attribute named *fx:factory* whose purpose is to create instances of objects whose classes do not have a default constructor [13]0. The attribute value, (see *observableArrayList* in Figure 5-12) is the name of a static, factory method which is responsible for creating instances of these objects [13].




```
<ComboBox>
    <items>
       <FXCollections fx:factory="observableArrayList">
            <String fx:value="Item 1" />
            <String fx:value="Item 2" />
            <String fx:value="Item 3" />
       </FXCollections>
    </items>
    <prefWidth>
        <Double fx:value="200.0" />
     </prefWidth>
</ComboBox>
```

Figure 5-12: FXCollections collection.

In order to deal with FXCollections, the algorithm checks whether the invoking a property of a node returns an object of type ObservableListWrapper that has a setter method.

5.2.4 Support for immutable objects

Immutable objects refer to those objects that do not have any setter method, thus is *not* possible to change the state of the object once it has been constructed. Builders that are mutable helper classes [13] are utilized in order to create instances of immutable types.

An example of an immutable object in JavaFX is the *Color* object (depicted in Figure 5-13) whose instances are constructed using the *ColorBuilder* mutable helper class.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.BorderPane?>
<?import javafx.scene.layout.Pane?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.paint.Color?>
<?import java.lang.String?>
<?import java.lang.Double?>

<BorderPane xmlns:fx="http://javafx.com/fxml">
   <top>
      <Pane>
         <children>
             <Label>
                 <id>
                   <String fx:value="lblTitle" />
                 </id>
                 <text>
                    <String fx:value="hi" />
                 </text>
                 <textFill>
                    <Color>
                       <red>
                          <Double fx:value="0.5" />
                       </red>
                       <blue>
                           <Double fx:value="1.0" />
                       </blue>
                       <green>
                            <Double fx:value="1.0" />
                        </green>
                    </Color>
                 </textFill>
             </Label>
         </children>
       /Pane>
     </top>
</BorderPane>
```

Figure 5-13: Example of immutable objects.

A default builder factory is provided by JavaFX called *JavaFX BuilderFactory* which is responsible for creating builders. However, a builder factory has been developed for the Ericsson visualization platform that supports both the instantiation of custom immutable components and the default JavaFX immutable types.

Immutable instances are constructed by invoking the `build()` method of a builder class instance (if any builder is provided).

Whenever a new node is visited, the algorithm checks if it has a builder. If this is the case, then the next step would be to determine which properties of the immutable object should be written to the DOM. Among all immutable object properties, only those included in the corresponding builder class should be considered since these are the properties for which it is possible to set the value for the immutable object property. However, it should be emphasized that not all builder class's properties should be serialized, rather only those that are set to a non-default value. Reflection is used to get the value of the builder class fields in order to compare them to the value of the corresponding node property.

5.2.5 Support for static properties

An FXML element might represent a static property. Unlike other FXML property elements that belong to the class instance for which they are defined, static properties "are not intrinsic to the class to which they are applied" [13]. They actually belong to the parent container of the node to which they are applied. Considering the example shown in Figure 5-14, three static properties named *rowIndex*, *columnIndex,* and *hgrow* have been defined

inside the *Label* instance element. However, these static properties do not originally belong to the Label instance, but rather hey are static methods of the Label's parent container (i.e. *GridPane*). That is why they are prefixed with the actual class name.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.GridPane?>
<?import java.lang.Integer?>
<?import javafx.scene.layout.Priority?>
<?import javafx.scene.control.Label?>
<?import java.lang.String?>

<GridPane xmlns:fx="http://javafx.com/fxml">
       <children>
              <Label>
                     <text>
                           <String fx:value="myLabel" />
                     </text>
                     <GridPane.rowIndex>
                           <Integer fx:value="10" />
                     </GridPane.rowIndex>
                     <GridPane.columnIndex>
                           <Integer fx:value="5" />
                     </GridPane.columnIndex>
                     <GridPane.hgrow>
                           <String fx:value="ALWAYS" />
                     </GridPane.hgrow>
              </Label>
       </children>
</GridPane>
```

Figure 5-14: Static propeties.

For instance, as shown the code snippet in Figure 5-14, the GridPane.rowIndex property is used to specify the row number of a Label instance within its parent container (i.e. GridPane).

To provide functionality to support static properties in the serialized FXML files, the algorithm first checks if a node has a parent once it is visited. If this is the case, then it looks through every *static getter* method of the parent node using reflection and then invokes all these methods on the currently visited node one by one. If method invocation returns a non-default value, then the static property should be written to the DOM in order to be subsequently included in the FXML document. Hence, the exact property element name (e.g. GridPane.rowIndex) should be derived from the static method name (`getRowIndex`) as well as the parent container name (GridPane). This is how the serialization function deals with static properties.

5.2.6 Support for fx:controller

If any controller class is provided for the FXML document, then it is passed to the serialization function as an input argument (along with the scene graph root node). The algorithm checks for the controller class. If one is provided, then this controller class is added as the FXML root node attribute named *fx:controller* (in keeping with the FXML syntax and semantics). The value of this property refers to the fully-qualified name of the controller class which is obtained using the Reflection API. Figure 5-15 depicts an FXML document including the *fx:controller* attribute for the root node in order to support the related controller class.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.Double?>
<?import java.lang.String?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.text.Text?>

<AnchorPane xmlns:fx="http://javafx.com/fxml"
    fx:controller="com.ericsson..fxml.ControllerClass">

       <children>
              <Text>
                     <text>
                           <String fx:value="java-Buddy" />
                     </text>
              </Text>
              <Label>
                     <text>
                           <String fx:value="hi" />
                     </text>
              </Label>
       </children>
</AnchorPane>
```

Figure 5-15: fx:controller attribute.

6 Results

The figures in this chapter depict serialization or verification results obtained when following the approach described in Chapter 3.

6.1 Verification of the scene graph traversal, changed property detection and FXML generation

Figure 6-1 illustrates the depth-first traversal (described in section 5.1.1) of the scene graph corresponding to the included FXML document which is the expected result of the serialization. The text beside each arrow corresponds to the property name, while numbers indicate the expected iteration order of the nodes. Figure 6-2 shows the generated FXML document. This example is used to verify that only the properties center, items, prefHeight, side, tabs, and text are changed.

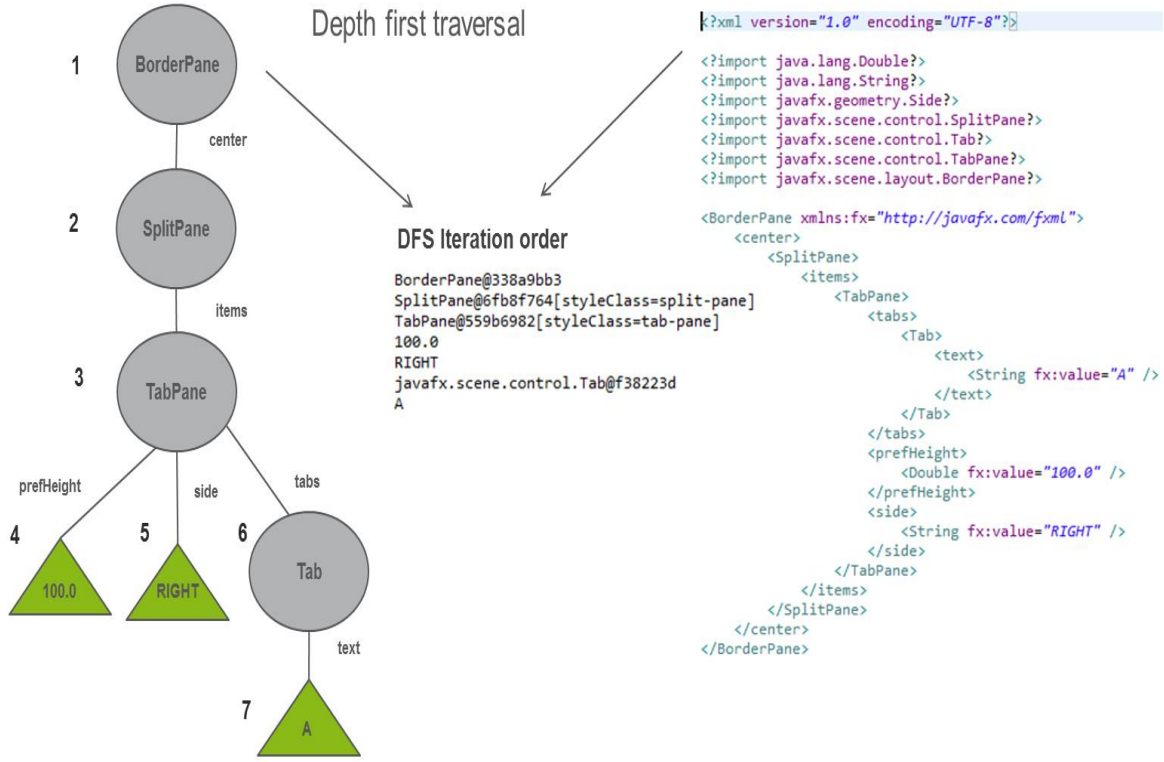


Figure 6-1: Scene graph depth-first traversal.

```
<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.Double?>
<?import java.lang.String?>
<?import javafx.geometry.Side?>
<?import javafx.scene.control.SplitPane?>
<?import javafx.scene.control.Tab?>
<?import javafx.scene.control.TabPane?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane xmlns:fx="http://javafx.com/fxml">
  <center>
    <SplitPane>
      <items>
        <TabPane>
          <prefHeight>
            <Double fx:value="100.0"/>
          </prefHeight>
          <side>
            <String fx:value="RIGHT"/>
          </side>
          <tabs>
            <Tab>
              <text>
                <String fx:value="A"/>
              </text>
            </Tab>
          </tabs>
        </TabPane>
      </items>
    </SplitPane>
  </center>
</BorderPane>
```

Figure 6-2: Changed property detection and FXML generation.

As will be shown in Figure 6-11 the serialization results have been verified against the expected FXML documents.

6.2 Verification of atomic property types

Figure 6-3 shows an FXML document and the corresponding serialized FXML including atomic property types. Instances of *primitive wrapper types* and/or *String* are represented as elements in the FXML file and the string representation of the atomic property value constitute the value of the fx:value attribute. It is obvious that the property order in the expected and the serialized FXML files differs due to the alphabetical order of the bean property introspection. Figure 6-11 also illustrates the evaluation/test result of the serialized FXML.

Expected FXML

```
<?xml version="1.0" encoding="UTF-
8"?>
<?import java.lang.String?>
<?import
javafx.scene.control.Label?>
<?importjavafx.scene.layout.BorderPa
ne?>
<?import java.lang.Double?>

<BorderPane
xmlns:fx="http://javafx.com/fxml">

<prefWidth>
  <Double fx:value="320.0" />
</prefWidth>
 <left>
   <Label>
    <text>
      <String fx:value="hi" />
    </text>
   </Label>
</left>
<bottom>
   <Label>
     <text>
      <String fx:value="myLabel" />
     </text>
   </Label>
</bottom>
</BorderPane>
```

Serialized FXML

```
<?xml version="1.0"encoding="UTF-
8"?>
<?import java.lang.Double?>
<?import java.lang.String?>
<?import
javafx.scene.control.Label?>
<?import
javafx.scene.layout.BorderPane?>

<BorderPane
xmlns:fx="http://javafx.com/fxml">
<bottom>
 <Label>
   <text>
     <String fx:value="myLabel"/>
   </text>
 </Label>
 </bottom>
 <left>
   <Label>
     <text>
       <String fx:value="hi"/>
     </text>
   </Label>
</left>
<prefWidth>
    <Double fx:value="320.0"/>
  </prefWidth>
</BorderPane>
```

Figure 6-3: Serialized primitives or atomic properties.

6.3 Verification of import statements

Figure 6-4 depicts import processing instructions of an FXML file together with the resulting serialized processing instructions. We emphasize that the order of the generated import statements might differ from the original order, but this does not affect the verification result as shown in Figure 6-11.

Expected FXML

```
<?xml version="1.0" encoding="UTF-
8"?>

<?import
javafx.scene.layout.BorderPane?>
<?import javafx.scene.layout.Pane?>
<?import
javafx.scene.control.Label?>
<?import javafx.scene.paint.Color?>
<?import java.lang.String?>
<?import java.lang.Double?>
```

Serialized FXML

```
<?xml version="1.0" encoding="UTF-
8"?>
<?import java.lang.Double?>
<?import java.lang.String?>
<?import
javafx.scene.control.Label?>
<?import
javafx.scene.layout.BorderPane?>
<?import javafx.scene.layout.Pane?>
<?import javafx.scene.paint.Color?>
```

Figure 6-4: Serialized import statements.

6.4 Verification of collections and FXCollections

Examples of serialized FXCollections and collections are provided in

Figure 6-5 and Figure 6-6. The FXCollections collection example is related to a writable ObservableList containing ComboBox items while the second collection implies an ObservableList of MenuItem. Figure 6-11 shows the evaluation results of both collections.

Expexted FXML

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.HBox?>
<?import
javafx.scene.control.TextField?>
<?import
javafx.collections.FXCollections?>
<?import java.lang.Double?>
<?import
javafx.scene.control.ComboBox?>
<?import java.lang.String?>

<HBox xmlns:fx="http://javafx.com/fxml"
fx:controller="com.ericsson.fxml.Contro
llerClass">

 <children>
   <TextField>
     <prefHeight>
       <Double fx:value="200.0" />
       </prefHeight>
    </TextField>
    <ComboBox>
<items>
  <FXCollections
fx:factory="observableArrayList">

    <String fx:value="Item 1" />
    <String fx:value="Item 2" />
    <String fx:value="Item 3" />
</FXCollections>
</items>
<prefWidth>
  <Double fx:value="200.0" />
</prefWidth>
</ComboBox>
</children>
<prefWidth>
   <Double fx:value="30.0"/>
</prefWidth>

</HBox>
```

Serialized FXML

```
<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.Double?>
<?import java.lang.String?>
<?import
javafx.collections.FXCollections?>
<?import
javafx.scene.control.ComboBox?>
<?import
javafx.scene.control.TextField?>
<?import javafx.scene.layout.HBox?>

<HBox
xmlns:fx="http://javafx.com/fxml"
fx:controller="com.ericsson.fxml.Contr
ollerClass">
  <children>
    <TextField>
      <prefHeight>
        <Double fx:value="200.0"/>
      </prefHeight>
    </TextField>
    <ComboBox>
      <items>
        <FXCollections
fx:factory="observableArrayList">
         <String fx:value="Item 1"/>
         <String fx:value="Item 2"/>
         <String fx:value="Item 3"/>
        </FXCollections>
      </items>
      <prefWidth>
        <Double fx:value="200.0"/>
      </prefWidth>
    </ComboBox>
  </children>
  <prefWidth>
    <Double fx:value="30.0"/>
  </prefWidth>
</HBox>
```

Figure 6-5: Serialized FXCollections collection.

Expected FXML

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.Double?>
<?import java.lang.String?>
<?import javafx.scene.control.Menu?>
<?import javafx.scene.control.MenuBar?>
<?import
javafx.scene.control.MenuItem?>
<?import javafx.scene.layout.VBox?>


<VBox
xmlns:fx="http://javafx.com/fxml">
  <children>
    <MenuBar>
      <id>
        <String fx:value="menuBar" />
      </id>
      <menus>
        <Menu>
         <text>
           <String fx:value="File" />
         </text>
         <items>
          <MenuItem>
           <text>
            <String fx:value="New" />
           </text>
          </MenuItem>
          <MenuItem>
           <text>
           <String fx:value="Open" />
           </text>
          </MenuItem>
          <MenuItem>
           <text>
           <String fx:value="Save" />
          </text>
          </MenuItem>
         </items>
      </Menu>
    </menus>
  </MenuBar>
</children>
<id>
 <String fx:value="vbox" />
</id>
<prefHeight>
    <Double fx:value="400.0" />
</prefHeight>
<prefWidth>
<Double fx:value="800.0" />
</prefWidth>
</VBox>
```

Serialized FXML

```
<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.Double?>
<?import java.lang.String?>
<?import javafx.scene.control.Menu?>
<?import javafx.scene.control.MenuBar?>
<?import javafx.scene.control.MenuItem?>
<?import javafx.scene.layout.VBox?>


<Vbox xmlns:fx="http://javafx.com/fxml">
  <children>
    <MenuBar>
      <id>
        <String fx:value="menuBar"/>
      </id>
      <menus>
       <Menu>
        <items>
         <MenuItem>
           <text>
             <String fx:value="New"/>
           </text>
         </MenuItem>
         <MenuItem>
           <text>
            <String fx:value="Open"/>
           </text>
         </MenuItem>
         <MenuItem>
           <text>
            <String fx:value="Save"/>
           </text>
         </MenuItem>
       </items>
          <text>
            <String fx:value="File"/>
          </text>
        </Menu>
      </menus>
    </MenuBar>
  </children>
  <id>
    <String fx:value="vbox"/>
  </id>
  <prefHeight>
    <Double fx:value="400.0"/>
  </prefHeight>
  <prefWidth>
    <Double fx:value="800.0"/>
  </prefWidth>
</VBox>
```

Figure 6-6: Serialized collection of MenuItem objects.

6.5 Verification of immutable objects

Color is an example of an immutable object composed of different properties. Among these, the original FXML is chosen to have blue, red, and green properties for the Color instance which is serialized as depicted in Figure 6-7. The serialized FXML document is also evaluated as shown in Figure 6-11.

Expected FXML

```
<?xml version="1.0" encoding="UTF-8"?>

<?import
javafx.scene.layout.BorderPane?>
<?import javafx.scene.layout.Pane?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.paint.Color?>
<?import java.lang.String?>
<?import java.lang.Double?>

<BorderPane
xmlns:fx="http://javafx.com/fxml">

<top>
 <Pane>
  <children>
   <Label>
    <id>
    <String fx:value="lblTitle" />
   </id>
   <text>
     <String fx:value="hi" />
   </text>
  <layoutX>
     <Double fx:value="14.0" />
  </layoutX>
  <layoutY>
     <Double fx:value="15.0" />
  </layoutY>
  <textFill>
    <Color>
      <red>
        <Double fx:value="0.5" />
      </red>
      <blue>
        <Double fx:value="1.0" />
      </blue>
      <green>
        <Double fx:value="1.0" />
      </green>
     </Color>
  </textFill>
 </Label>
</children>
</Pane>
</top>
</BorderPane>
```

Serialized FXML

```
<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.Double?>
<?import java.lang.String?>
<?import javafx.scene.control.Label?>
<?import
javafx.scene.layout.BorderPane?>
<?import javafx.scene.layout.Pane?>
<?import javafx.scene.paint.Color?>

<BorderPane
xmlns:fx="http://javafx.com/fxml">
 <top>
  <Pane>
   <children>
    <Label>
     <id>
      <String fx:value="lblTitle"/>
     </id>
     <layoutX>
       <Double fx:value="14.0"/>
     </layoutX>
     <layoutY>
       <Double fx:value="15.0"/>
     </layoutY>
     <text>
        <String fx:value="hi"/>
     </text>
     <textFill>
        <Color>
         <blue>
           <Double fx:value="1.0"/>
         </blue>
         <green>
           <Double fx:value="1.0"/>
         </green>
         <red>
           <Double fx:value="0.5"/>
         </red>
        </Color>
    </textFill>
   </Label>
 </children>
</Pane>
</top>
</BorderPane>
```

Figure 6-7: Serialized immutable object.

6.6 Verification of static properties

As it is shown in Figure 6-8, it is obvious that the serialized FXML presents the static properties in a different order than the original FXML. See Figure 6-11 for the evaluation result.

Expected FXML

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.GridPane?>
<?import java.lang.Integer?>
<?import javafx.scene.layout.Priority?>
<?import javafx.scene.control.Label?>
<?import java.lang.String?>

<GridPane
xmlns:fx="http://javafx.com/fxml">

<children>
  <Label>
   <text>
     <String fx:value="myLabel" />
   </text>
   <GridPane.rowIndex>
      <Integer fx:value="10" />
   </GridPane.rowIndex>
   <GridPane.columnIndex>
      <Integer fx:value="5" />
   </GridPane.columnIndex>
   <GridPane.hgrow>
       <String fx:value="ALWAYS" />
   </GridPane.hgrow>
  </Label>
</children>
</GridPane>
```

Serialized FXML

```
<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.Integer?>
<?import java.lang.String?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.layout.Priority?>

<GridPane
xmlns:fx="http://javafx.com/fxml">

  <children>
    <Label>
      <GridPane.columnIndex>
        <Integer fx:value="5"/>
      </GridPane.columnIndex>
      <GridPane.rowIndex>
        <Integer fx:value="10"/>
      </GridPane.rowIndex>
      <GridPane.hgrow>
        <String fx:value="ALWAYS"/>
      </GridPane.hgrow>
      <text>
       <String fx:value="myLabel"/>
      </text>
    </Label>
  </children>
</GridPane>
```

Figure 6-8: Serialized static properties.

6.7 Verification of fx:controller attribute

As shown in Figure 6-9, the related controller class attached to the original FXML file is also added to the serialized one and the result of the evaluation is depicted inFigure 6-11.

Expected FXML

```
<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.Double?>
<?import java.lang.String?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import
javafx.scene.control.TextField?>
<?import
javafx.scene.layout.AnchorPane?>
<?import javafx.scene.text.Text?>

<AnchorPane
xmlns:fx="http://javafx.com/fxml"
fx:controller="com.ericsson.fxml.Control
lerClass">

 <children>
   <Text>
    <text>
     <String fx:value="java-Buddy" />
    </text>
   </Text>
   <Label>
     <text>
        <String fx:value="hi" />
     </text>
   </Label>
   <TextField>
     <id>
      <String fx:value="textField" />
     </id>
  </TextField>
  <Button>
    <id>
     <String fx:value="myButton" />
    </id>
    <text>
     <String fx:value="Click me!!" />
   </text>
 </Button>
 <Label>
    <id>
     <String fx:value="label" />
    </id>
</Label>
</children>
<prefHeight>
     <Double fx:value="200.0" />
</prefHeight>
<prefWidth>
     <Double fx:value="400.0" />
</prefWidth>
</AnchorPane>
```

Serialized FXML

```
<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.Double?>
<?import java.lang.String?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import
javafx.scene.control.TextField?>
<?import
javafx.scene.layout.AnchorPane?>
<?import javafx.scene.text.Text?>

<AnchorPane
xmlns:fx="http://javafx.com/fxml"
fx:controller="com.ericsson.fxml.Contro
llerClass">
 <children>
  <Text>
   <text>
    <String fx:value="java-Buddy"/>
   </text>
  </Text>
  <Label>
   <text>
     <String fx:value="hi"/>
   </text>
  </Label>
  <TextField>
   <id>
     <String fx:value="textField"/>
   </id>
  </TextField>
  <Button>
   <id>
      <String fx:value="myButton"/>
   </id>
   <text>
    <String fx:value="Click me!!"/>
   </text>
 </Button>
 <Label>
    <id>
        <String fx:value="label"/>
    </id>
 </Label>
</children>
<prefHeight>
    <Double fx:value="200.0"/>
</prefHeight>
<prefWidth>
    <Double fx:value="400.0"/>
</prefWidth>
</AnchorPane>
```

Figure 6-9: The fx:controller attribute for the controller class.

6.8 Verification of Enumerations

Enumerations are treated as the *String* instance for the FXML element name and the enumeration value for the value of fx:value attribute as illustrated in Figure 6-10 and verified in Figure 6-11.

Expected FXML

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.Double?>
<?import java.lang.String?>
<?import javafx.geometry.Side?>
<?import
javafx.scene.control.SplitPane?>
<?import javafx.scene.control.Tab?>
<?import javafx.scene.control.TabPane?>
<?import
javafx.scene.layout.BorderPane?>

<BorderPane
xmlns:fx="http://javafx.com/fxml">
 <center>
  <SplitPane>
    <items>
     <TabPane>
      <tabs>
         <Tab>
          <text>
            <String fx:value="A" />
          </text>
         </Tab>
      </tabs>
      <prefHeight>
       <Double fx:value="100.0" />
      </prefHeight>
      <side>
        <String fx:value="RIGHT" />
      </side>
    </TabPane>
   </items>
  </SplitPane>
 </center>
</BorderPane>
```

Serialized FXML

```
<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.Double?>
<?import java.lang.String?>
<?import javafx.geometry.Side?>
<?import
javafx.scene.control.SplitPane?>
<?import javafx.scene.control.Tab?>
<?import javafx.scene.control.TabPane?>
<?import
javafx.scene.layout.BorderPane?>

<BorderPane
xmlns:fx="http://javafx.com/fxml">
 <center>
  <SplitPane>
   <items>
    <TabPane>
      <prefHeight>
        <Double fx:value="100.0"/>
      </prefHeight>
      <side>
        <String fx:value="RIGHT"/>
      </side>
      <tabs>
        <Tab>
         <text>
            <String fx:value="A"/>
         </text>
         </Tab>
       </tabs>
    </TabPane>
   </items>
  </SplitPane>
 </center>
</BorderPane>
```

Figure 6-10: Serialized enumeration.

6.9 Test case verification

A summary of all unit tests used for verification is shown in Figure 6-11. Each test verifies a particular serialized FXML file.

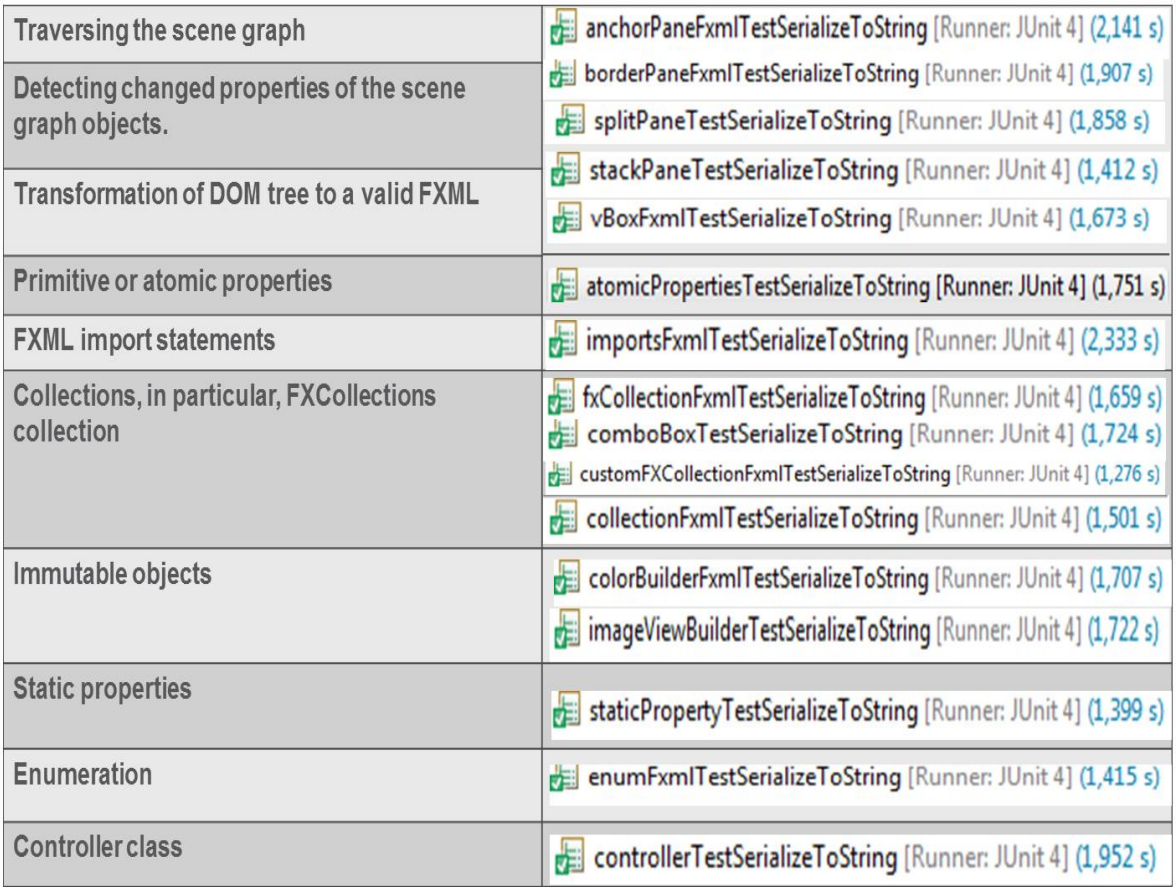


Figure 6-11: JUnit test results for the serialized FXML files. (Note that the test execution times are given with a comma as the decimal point and a total of approximately 500 test cases exist in the test folder.)

6.10 Code coverage analysis

The statistics shown in Table 6-1 indicate the percentage of code coverage for FXMLWriter.java (the main class that contains the serialization function and its dependencies). Of the 1217 instructions for FXMLWriter.java, 82 instructions are evidently not executed. They are mainly related to the different categories of nodes that have been considered in order to find the edges between the scene graph immutable objects/nodes (those that need a builder for construction).

Table 6-1: Code coverage percentage using EclEmma.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
FXMLWriter.java	93.3%	1135	82	1217

6.11 Usability and reliability of the FXML serilization function

This section summarizes the responses to the questionnaire in Appendix A. These responses are presented in Appendix B. The answers were provided by three research engineers at Ericsson who work with the visualization platform on a daily basis.

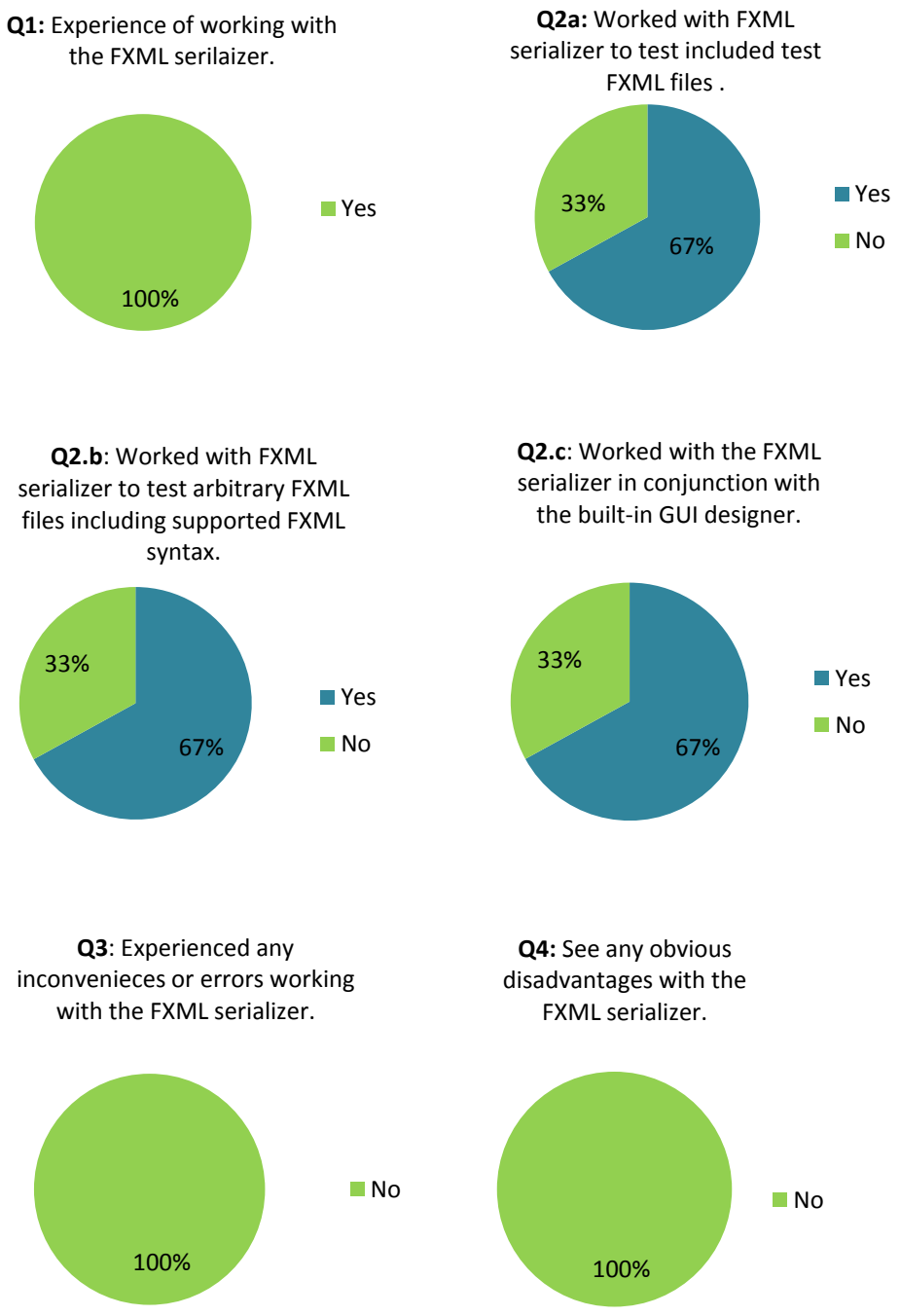


Figure 6-12: Summary of questionnaire results regarding FXML serializer usability and reliability.

As can be seen in Figure 6-12, all the participants in the survey had experience working with or testing the FXML serializer. The collected responses for three sections of the second question have been quantified to be either yes or not in order to understand the particular aspect of the FXML serializer that has been tested by the users (i.e., included test FXML files, any arbitrary FXML files, or the FXML files in conjunction with the built-in GUI designer). The results from questions 3 and 4 show that no user experienced any inconveniece or errors (or disadvantages) working with the serialization function.

7 Evaluation and Conclusions

The initial idea of this thesis was to provide a serialization function so that the integration of this function with the results of a parallel thesis project (that provides a built-in designer for the layout of the visualization platform) would reduce the amount of work required to construct a new demonstrator based on the visualization platform The general goal was divided into three sub-goals as described in section 1.3. All of the sub-goals were achieved during this project and will be elaborated on with regard to the evaluation methods in the subsequent sections of this chapter.

7.1 Evaluation of possible serialization frameworks

The results of the investigation and theoretical studies revealed a set of APIs and frameworks that were potential candidates to aid in the development of a serialization algorithm. The evaluation result revealed that the preferable APIs were those that require less implementation effort compared to the equivalent alternatives and that are already included in the standard Java package (i.e. Java Reflection API, Java Introspection API, and JAXP).

7.2 Evaluation of designed algorithm

The test results presented in the Chapter 6 were used to evaluate the proposed algorithm against the requirements (in order to fulfill the third sub-goal of the thesis presented in section 1.3). It should be mentioned that a successfully passed unit test is not necessarily an indication of the fulfillment of the corresponding requirement, but rather only indicates that the serialized FXML conforms to the original one in the sense that they are similar. Hence, in order to make sure that the requirements are actually fulfilled, these tests should be carefully designed. The tests presented in Chapter 6 were evaluated with regard to the requirements and the serialization results were also evaluated. These evaluations are given below.

7.2.1 Evaluation of scene graph traversal, changed properties detection and FXML generation

In order to evaluate the design decision for the scene graph traversal, changed properties detection, and valid FXML generation, the scene graph presented in Figure 6-1 is considered. This scene graph is composed of different leaf nodes and branch nodes of different Java types and various bean properties.

The iteration order shown in Figure 6-1, verifies the depth-first traversal. Despite different orders of TabPane properties in the test FXML, a similar FXML is generated as depicted in Figure 6-2. The successfully passed unit test (shown in Figure 6-11: JUnit test results for the serialized FXML files.) is an indication of the requirement being fulfilled. It can be seen that in order to verify the traversal, changed property detection, and valid FXML generation, various FXML files have been considered including the most common layout containers of the scene graph (e.g. AnchorPane, BorderPane, StackPane, VBox, etc.). The reason for explicitly considering each of them is that each pane class has different characteristics and properties. Hence, evaluating/testing a variety of containers ensures that the designed solution works properly for the most commonly used layout panes. The algorithm will also process the other Pane classes. However, there maybe corer case problems as will be described in Section 7.2.9.

7.2.2 Evaluation of Atomic property types

In order to evaluate the proposed solution for the atomic properties of a scene graph, the test presented in Figure 6-3 Figure 6-3: Serialized primitives or atomic properties.was considered. This test case includes instances of String and Double (which are primitive wrapper types). The corresponding unit test was passed as shown in Figure 6-11. This verifies that this requirement is met. However, atomic properties are not limited to this particular test as they are included in multiple FXML files used for verification as presented in Chapter 6.

7.2.3 Evaluation of Import statements

Figure 6-4 depicts the test of the import document to evaluate the solution's ability to handle the import statements of the FXML document as well as the serialized import document. The generated import document consists of all of the different fully-qualified class names of those classes used in the body of the FXML document.

It is clear that the import statement order is different in the resulting FXML compared to the original one due to alphabetical order of bean property introspection. Furthermore, the algorithm sorts the fully-qualified class name of every visited node alphabetically. However, the result of the unit testing verifies the similarity of FXML documents (See Figure 6-11 for the FXML import statements verification result). It should be mentioned that import statements are also evaluated for every designed test FXML. Hence, this requirement is also fulfilled.

7.2.4 Evaluation of Collections and FXCollections

In order to evaluate the proposed solution for FXCollections, the test FXML file should contain any control whose items populate a writable ObservableList. Different FXML files containing instances of ListView, ComboBox, TableView, etc. that test these requirements have been considered. The test items were either of a custom type or predefined Java types, such as String.

Among the previously mentioned controls, ComboBox is specifically included in the test FXML (as presented in
Figure 6-5). It can be seen in this earlier example that an FXCollections instance is used to populate the writable ObservableList of String type. Hence the unit test was successfully passed as depicted in Figure 6-11 which is an indication of correct support for the FXCollections requirement.

To evaluate the design decision for collections, the test FXML document should be an FXML file that contains a JavaFX control whose items populate an ObservableList that has no setter, such as Menu presented in Figure 6-6. In this example, Items of Menu populate an ObservableList of MenuItem. The successful verification of the serialized FXML for this case (see Figure 6-11 for collections) implies that this requirement is also met.

7.2.5 Evaluation of Immutable objects

To evaluate the proposed solution for immutable objects, the tests should include an immutable instance with any of its properties set to a value different from the default value (which is also the case for any property included in the test FXML documents). The immutable instance could be a custom type which is already defined by the visualization tool or any of the default JavaFX immutable types.

For the immutable Color instance defined in the test FXML which was presented in Figure 6-7, three properties (red, green, and blue) have been defined and set to values other than the default value (which is 0.0). The result is that a similar FXML document is produced with a different order for the properties of this Color instance. The verification result was shown in

Figure 6-11. The verification implies the fulfillment of the requirement for immutable objects, such as Color and ImageView.

7.2.6 Evaluation of Static properties

In order to evaluate the design decision for the static properties, the test FXML should contain a parent container that any of its static methods which is prefixed with the parent container name is considered as the property element of the child element which is set to a value. Figure 6-8 presents a GridPane parent container including three static methods named `rowIndex`, `columnIndex`, and `hgrow` that are invoked on the *Label* instance to specify the position of Label instance inside the *GridPane* parent container.

The generated FXML (shown in Figure 6-8) is similar to the original one. This case is verified in Figure 6-11, indicating that static properties feature is also supported according to the test FXML which was designed based on the static properties definition.

7.2.7 Evaluation of fx:controller

The only thing to do in order to evaluate the proposed solution's ability to support the fx:controller, is to attach a controller class to any FXML test file and set the fully-qualified controller class name to the fx:controller attribute value as illustrated in Figure 6-9. Figure 6-11 shows the verification result for fx:controller which implies the fulfillment of the fx:controller requirement.

7.2.8 Evaluation of Enumerations

All enumerations extend the java.lang.Enum class and have a number of constant values with string representations. Hence, to evaluate the solution provided for enumerations, a test FXML file is used that contains properties whose getter method invocation returns an object that extends the Enum class. The FXML file shown in Figure 6-10 contains a *side* property whose getter invocation on the *TabPane* instance returns an object of *Side* type which in turn extends the Enum class. Moreover, the value of the property is the string representation of a constant value (e.g. RIGHT).

The serialization algorithm produces an FXML file similar to the original one (as depicted in Figure 6-10). The verification results presented in Figure 6-11 for enumerations also indicates the fulfillment of this requirement.

In conclusion, the algorithm proposed in this thesis (as presented in Chapter 2) enables Java object/bean serialization to the FXML representation. Specificaly, the algorithm shows that using built-in Java functionality, it is possible to serialize a JavaFX scene graph to the FXML text format based on the requirements list of features presented in Chapter 4, provided that the tests are carefully designed. It is obvious that the property order and the import statements order in the generated serialized FXML file may differ from the original one due to the alphabetical order of the bean property introspection. However, this is not an issue for verification since XMLUnit is able to verify that two FXML files are similar. However, some limitations have been encountered during this work that will be described in the next section.

7.2.9 Identified limitations of the algorithm

The first identified limitation in the algorithm is that the information required to generate the FXML document might not always be extracted from the scene graph, which in turn makes some FXML syntax and semantics one-way functions with regard to the scene graph object introspection.

What is meant by “one-way function” is that, it is not feasible to extract the entire FXML syntax and semantics by reading and analysing the scene graph, although it is possible to generate an FXML file that corresponds to the same application layout as the original one.

For example, two references that point to the same object, could be specified in the FXML by using <fx:reference> element. However, when examining the scene graph and becoming aware that two objects have the same reference, they need not have been written as <fx: reference> in the original FXML. For this reason the serialized FXML might represent the same user interface as the original one, although they might be textually different. This in turn could be an issue with regard to the evaluation. The same is true for fx:copy, fx:define, and fx:id. All of these features should be considered in future work. Regarding the <fx:reference> it should also be emphasized that a reference comparison could be problematic since the `equals()` method can be overridden in Java, which leads to an object state comparison rather than a reference comparison.

Secondly, the serialization function has been tested and evaluated by using different FXML files that contain various objects and properties in order to ensure that the algorithm works correctly for its intended purpose. However, the evaluation results revealed that corner cases exist. These corner cases were identified during detection of the scene graph object’s changed properties, hence the value of some properties were considered as the non-default value even though they had the default value.

The eventDispatcher and dividerPositions are two properties that the algorithm treats as the corner cases. Getter method invocation of the eventDispatcher property returns a value of EventDispatcher type which specifies the event dispatcher for the node and getter method invocation of the dividerPositions property returns an array of double[ ] containing the position of each divider. The returned value and the default value for both of these properties are equal. However, the default object id has a value different from the actual object id. For this reason these properties are treated as non-default properties. Since both properties are counted as the standard property (as described in section 5.1.2), for every standard property the algorithm first checks the property name against the name of these two properties. If they are equal, the algorithm does *not* add the property value to the children list in order to avoid the default value being written to the FXML document.

Note that the test FXML files should never include any properties with a default value otherwise the test will fail.

If I had to do this project again, I would have spent less time investigating different XML serialization frameworks and instead, provide features that are currently left as future work (see Section 7.4). However, it was necessary to investigate different APIs in order to compare them from different perspectives.

7.3 Evaluation of the questionnaire considering usability and reliability of the provided functionality

According to the questionnaire results presented in Figure 6-12, it is evident that the three users generally had a positive attitude towards the serialization function. This conclusion is based on the fact that all the users have experience working with the FXML serializer and none of them have noted any disadvantages with regard to the serializer or the set of FXML syntax that it supports.

When it comes to the approach by which the users tested the serializer in question 2, it can be seen that each of the included approaches (i.e. testing with the included test FXML file, testing with an arbitrary FXML file that contains supported FXML syntax, and testing in

conjunction with the built-in GUI designer) was examined by 67% of the users (two out of three users).

Considering the results from questions 3 and 4 that seek to identify if any disadvantages or inconveniences were encountered, no disadvantages or inconveniences has been mentioned for the provided function. According to their responses, the FXML serializer works fine (both in isolation and in conjunction with the built-in GUI desinger) as long as the test FXML files include all of the supported FXML constructs. However, if the FXML contains syntax that is not currently supported by the function (since the FXML generated by the WYS/WYG run-time GUI editor is out of the scope of the thesis), the serializer fails. All the respondents state that there needs to be extensions to the function in the future work in order to support FXML constructs that are currently out of the scope of this project in order to make it capable of handling all the visualization platform's classes.

7.4 Future work

Due to the limited duration of this thesis project, it was not possible to construct an algorithm that supports serialization of the complete FXML specification. Hence, the following features remain to be provided in order to have a complete FXML serializer and these features are considered as future work or extensions to the work conducted as part of this master's thesis project:

- **Generating fx:define**: This is utilized in order to define objects or variables that are not included in the FXML object hierarchy. Since it is cannot be extracted from the scene graph (because it is outside of the object hierarchy), in this thesis project it has been regarded as a one-way function.
- Performing event handler serialization might be one of the trickiest parts to implement as this handler does not exist in the scene node.
- **Location resolution**: Location resolution might violate the property element design consideration since sometimes it is required to include a property attribute instead of a property element in the FXML. The URL attribute for Image instances is an example of this kind of property.
- **Variable resolution**: Variable resolution might also violate the property element design consideration that has been considered in the algorithm's design.
- **Resource resolution**: Without taking the advantage of the resource resolution, it is not possible to substitute instances of specified resources with their specified values. Resources are equivalent to variables in the FXML file.
- Escape sequence refers to adding a backslash character at the beginning of an attribute value in order to escape resource resolution prefixes such as # and $ by which the attribute value is started and it is actually part of the attribute value.
- **Expression bindings**: Without providing the binding feature for FXML, when the value of a variable changes, the change is not reflected in the property/attribute value to which the variable value is assigned.
- **Preserving fx**: **include relations**: Without support for this feature it is impossible to embed another FXML document within an FXML file.
- **Support for fx:root**: The <fx:root> element makes it possible to refer to a previously defined root element.

- **Support for fx:script**: Using <fx:script> it is possible to embed script code within the FXML document. However, it is not possible to extract such a script from the scene node.
- Support for ObservableMap
- **Support for fx:reference**: Suport for fx:reference is used to create a new reference to an existing element. By using an <fx:reference> the amount of FXML code is reduced, especially when it is necessary to refer to the same object multiple times. However, as discussed previously, the <fx:reference> is a one-way function and also might be problematic when it comes to the reference comparison if the `equals()` method is overridden.
- **Support for fx:copy**: fx:copy is utilized in order to create a copy of an object, but this is also a one-way function.
- **Support for fx:id**: As described previously, this is a one-way function. Fx:id is used in order to inject FXML elements into the corresponding controller class. Without using the fx:id attribute, it is impossible to attach the application logic to the application layout in order to handle events.

In summary, in order to serialize a scene graph object to an FXML document that perfectly maps to the FXML syntax and semantics and can be successfully verified, it must be possible to extract the required information from the scene graph using object introspection. This does not happen when the FXML document is included the above one-way functions. In my opinion, construction of serialization for the complete FXML specification might be problematic even if it is possible with regard to these one-way functions.

7.5 Required reflections

This master's thesis project facilitates construction of a new visualization application by providing a serialization function particularly designed to save the layout of the visualization application which is constructed using WYS/WYG run-time GUI editor (developed in a parallel thesis project). Providing the "save" function for constructing applications based on the visualization platform, improves the user's experience and leads to increased user satisfaction, thus contributing to a positive **social** effect of this master's thesis project.

Constructing the application layout (FXML file) by hand would be costly in terms of the required effort and time. However, using the serialization function, the amount of time and work is reduced which leads to a positive **economic** contribution of this master's thesis project.

For **ethical** reasons, the name of the visualization platform was not disclosed during this master's thesis project and the test application depicted in Figure 2-1 on page 9 was generated based on the fake data and not actual simulation results. Furthermore, the names of Ericsson internally supplied documents were not revealed in the bibliography.

Bibliography

[1] A. Williamson, 'XML Serialization of Java Objects', *Java Developers Journal*, 2004. [Online]. Available: http://www2.sys-con.com/itsg/virtualcd/java/archives/0806/milne/index.html . [Accessed: 20-February-2013].

[2] G. Denaro and L. Mariani, 'Towards testing and analysis of systems that use serialization', *Elsevier, Electronic Notes in Theoretical Computer Sciene*, vol. 116, pp. 171–184, 2005.

[3] S. D. Halloway, 'Serialization', in *Component Development for the Java Platform*, Addison-Wesley Professional, 2001, pp. 105–106.

[4] M. Hericko, M. B. Juric, I. Rozman, S. Beloglavec, and A. Zivkovic, 'Object serialization analysis and comparison in Java and .NET', *ACM SIGPLAN Notices*, vol. 38, no. 8, pp. 44–54, August 2003.

[5] J.-M. L. Goff, H. Stockinger, R. McClatchey, Z. Kovacs, P. Martin, N. Bhatti, and W. Hassan, 'Object Serialization and Deserialization Using XML'. 24-May-2001, Available at http://cms.web.cern.ch/search/node/Object%20Serialization%20and%20Deserialization%20Using%20XML#search-results-CDS .

[6] K. Maeda, 'Performance Evaluation of Object Serialization Libraries in XML, JSON and Binary Formats', presented at the The Second International Conference on Digital Information and Communication Technology and its Applications(DICTAP2012), Bangkok, Thailand, 2012, pp. 177 – 182, DOI:10.1109/DICTAP.2012.6215346.

[7] G. Imre, M. Kaszó, T. Levendovszky, and H. Charaf, 'A Novel Cost Model of XML Serialization', *Elsevier, Electronic Notes in Theoretical Computer Science*, vol. 261, pp. 147–162, 2010.

[8] J. Weaver, W. Gao, S. Chin, D. Iverson, and J. Vos, *Pro JavaFX 2: A Definitive Guide to Rich Clients with Java Technology*. Apress, 2012, ISBN: 1430268727, 978-1430268727.

[9] E. Dahlman, S. Parkvall, and J. Skold, 'Background of LTE', in *4G: LTE/LTE-Advanced for Mobile Broadband*, Academic Press, 2011, pp. 1–11.

[10] Oracle, 'JAVAFX 2.0 THE PREMIER PLATFORM FOR RICH ENTERPRISE CLIENT APPLICATIONS'. Oracle, 2011, Available at http://www.oracle.com/technetwork/java/javafx/overview/javafx-2-datasheet-496523.pdf .

[11] Ericsson Research, Ericsson confidential internal document. 2012.

[12] Oracle, 'JavaFX CSS Reference Guide', *JavaFX*, 2013-2008. [Online]. Available: http://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html. [Accessed: 05-February-2013].

[13] Oracle, 'Introduction to FXML | JavaFX 2.2', *JavaFX*, 21-June-2012. [Online]. Available: http://docs.oracle.com/javafx/2/api/javafx/fxml/doc-files/introduction_to_fxml.html. [Accessed: 06-February-2013].

[14] I. Fedortsova, 'JavaFX/Mastering FXML'. April-2013, Available at http://docs.oracle.com/javafx/2/fxml_get_started/jfxpub-fxml_get_started.pdf .

[15] C. Castillo, 'JavaFX JavaFX Architecture'. April-2013, Available at http://docs.oracle.com/javafx/2/architecture/jfxpub-architecture.pdf .

[16] G. Chappell, 'Getting Started with JavaFX'. April-2013, Available at http://docs.oracle.com/javafx/2/get_started/jfxpub-get_started.pdf .

[17] F. Zhao, 'The Algorithm Analyses and Design about the Subjective Test Online Basing on the DOM Tree', presented at the International Conference on Computer Science and

Software Engineering, Wuhan, Hubei, 2008, vol. 5, pp. 577 – 581, DOI:10.1109/CSSE.2008.57.
[18] J. Bloch, 'Effective Java', in *Effective Java*, 2nd ed., Addison-Wesley, 2008, pp. 73–80.
[19] P. Nierneyer and J. Knudsen, *Learning Java*, 3rd ed. O'Reilly Media, 2005, ISBN: 0596008732, 978-0596008734.
[20] J. Friesen, *Beginning Java 7*. Apress, 2011, ISBN: 1430239093 , 978-1430239093.
[21] 'Retrieving Class Objects', *The Java Tutorial*, 2013-1995. [Online]. Available: http://docs.oracle.com/javase/tutorial/reflect/class/classNew.html . [Accessed: 04-May-2013].
[22] I. R. Forman and N. Forman, *Java Reflection in Action*. Manning Publications, 2004, ISBN: 1932394184, 978-1932394184.
[23] 'Trail: The Reflection API (The Java™ Tutorials)', *The Java Tutorials*, 2013-1995. [Online]. Available: http://docs.oracle.com/javase/tutorial/reflect/ . [Accessed: 04-May-2013].
[24] J. Hunt, *Essential JavaBeans fast*. Springer, 1998, ISBN: 1852330325, 978-1852330323.
[25] 'Lesson: Introspection', *The Java Tutorials*, 2006-1995. [Online]. Available: http://geti.dcc.ufrj.br/cursos/fes_2008_1/javatutorial/javabeans/introspection/index.html . [Accessed: 05-May-2013].
[26] E. R. Harold, *Processing XML with Java: A Guide to SAX, DOM, JDOM, JAXP, and TrAX*. Addison-Wesley Professional, 2002, ISBN: 0201771861, 978-0201771862.
[27] Y. Xiang, B. Zhang, G. Li, and X. Li, 'The Application and Analysis of Netconf Subtree Filtering based on SAX and DOM', presented at the 2010 International Conference on Measuring Technology and Mechatronics Automation (ICMTMA 2010), Changsha, China, 2010, vol. 3, pp. 758 – 761, DOI:10.1109/ICMTMA.2010.562.
[28] Oracle, 'Extensible Stylesheet Language Transformations APIs (The Java™ Tutorials > Java API for XML Processing (JAXP) > Introduction to JAXP)', *The Java Tutorials*, 2013-1995. [Online]. Available: http://docs.oracle.com/javase/tutorial/jaxp/intro/extensible.html . [Accessed: 06-May-2013].
[29] B. McLaughlin, *Java & XML, 2nd Edition: Solutions to Real-World Problems*, 2nd ed. O'Reilly Media, 2001, ISBN: 0596001975 , 978-0596001971.
[30] 'XStream', *XStream*, 19-January-2013. [Online]. Available: http://xstream.codehaus.org/index.html. [Accessed: 09-May-2013].
[31] D. M. Sosnoski, 'JiBX: Bindings Tutorial', 2008-2003. [Online]. Available: http://jibx.sourceforge.net/tutorial/binding-tutorial.html. [Accessed: 10-May-2013].
[32] 'Castor 1.3.3 RC1 - Reference documentation', *Castor*, 2013. [Online]. Available: http://castor.codehaus.org/reference/1.3.3rc1/html-single/index.html. [Accessed: 19-May-2013].
[33] M. Bajorek and J.Nowak**,** 'The role of a mobile device in a home monitoring healthcare system', Proceedings of the Federated Conference on Computer Science and Information Systems, pp. 371–374, 2011
[34] E. M. Grinkrug, '3D Modeling by means of JavaBeans', Proceedings of the 12th international workshop on computer science and information technologies CSIT'2010, Moscow – Saint-Petersburg, Russia, 2010.
[35] F. Zhao, 'The Algorithm Analyses and Design about the subjective test online Basing on The DOM Tree', International Conference on Computer Science and Software Engineering, 2008
[36] A. Henver and S. Chatterjee, 'Design research in information Systems', Springer, 2010, ISBN: 1441956522, 978-1441956521.

[37] P. Johannesson and E. Perjons, ‘A Design science Primer’, 2012, ISBN: 1477593942, 978-1477593943
[38] ‘JUnit, A programmer-oriented testing framework for Java’ [Online] Available: http://junit.org/ [Accessed 10-July-2013]
[39] T. Bacon and S. Bodewig, ‘XMLUnit Java User’s Guide’ [Online] Available: http://xmlunit.sourceforge.net/userguide/html/ . [Accessed: 10-July-2013].
[40] [Online] Available: http://www.eclemma.org/ [Accessed: 14-July-2013].
[41] Daniel Jonsson, “Configuration and layout tools for visualization software”, Master’s thesis work in progress, October 2013.
[42] T. H. Cormen, ‘Introduction to Algorithms’, 2nd edition, 2001, pp. 540-545, ISBN: 0262032937, 9780262032933
[43] G. T. Heineman, G. Pollice, and S. Selkow, ‘Algorithms in a Nutshell’, O'Reilly Media, 2009, pp. 149-151, ISBN: 1449391133, 9781449391133
[44] B. Burke, ‘Restful Java with JAX-RS’, O'Reilly Media, 2010, pp. 77-80, ISBN: 144938305X, 978144938305

Appendix A

Questionnaire for evaluation of usability and reliability of the FXML serialization function.

1. Have you had any experience of working with the FXML serialization function provided in the FXMLWriter class?

○Yes
○No

2. You have worked with the serialization function:

□ In order to test the compliance for the provided serialization function for the included test FXML files.
□ In order to test the compliance for the provided serialization function for any arbitrary test FXML files that includes FXML syntax that are supported by the serialization function.
□ In conjunction with the built-in GUI designer in order to save the layout of the application.

3. Have you experienced any inconveniences or errors working with the FXML serializer? If yes, shortly describe what confuses you or what causes the error(s)?

4. Do you see any obvious disadvantages with the provided FXML serializer.

Appendix B

Summary of questionnaire results

A summary of the collected responses to the questionnaire in Appendix A is presented here. Three research engineers at Ericsson who work with the visualization platform on a daily basis have answered the questionnaire. Their responses were:

Response 1 of 3:

1. Yes
2. In order to test the compliance for the provided serialization function for any arbitrary test FXML files that includes FXML syntax that is supported by the serialization function.
   In conjunction with the built-in GUI designer in order to save the layout of the application.
3. No. Only that all classes of the system are currently not supported. I don't have the knowledge to tell what causes the errors, but there are problems with some custom classes for example.
4. No, but there is some future work that has to be done before incorporating it into the current system.

Response 2 of 3:

1. Yes
2. In order to test the compliance for the provided serialization function for the included test FXML files.
3. No, all test cases pass and the FXML test files can be serialized by the FXML Serializer.
4. The structure of FXML files can be flexible and very complex and to fully support serialization of all elements possible would require a lot of work and we consider it as out of scope for this thesis. The current serializer works well with the elements that are supported by the serializer.

Response 3 of 3:

1. Yes
2. In order to test the compliance for the provided serialization function for the included test FXML files.
   In order to test the compliance for the provided serialization function for any arbitrary test FXML files that includes FXML syntax that are supported by the serialization function.
   In conjunction with the built-in GUI designer in order to save the layout of the application.
3. Not within the scope of the thesis requirements. The complete FXML syntax was not included in the M.Sc thesis scope and hence I managed to make the serialization function fail for FXML constructs that are outside the scope of the functionality requested by Ericsson for this thesis. This was achieved by using FXML files containing constructs not supported by the FXML serialization function. In the future, Ericsson will have to extend the serialization function with support for all FXML construction in order for us to finalize the integration of the FxmlWriter and the built-in GUI designer.
4. Not any disadvantages, however it is unclear at the moment if the design is sufficient to cover the FXML constructs not included in the scope of the thesis. One example of such an issue could be that only element value serialization is supported in the

solution even though attribute value serialization is required to support FXML location resolution (Which was not included in the scope of this thesis).

TRITA-ICT-EX-2013:232

www.kth.se