Tampere University

Matti Panula

A DOMAIN SPECIFIC GRAPHICAL USER INTERFACE FRAMEWORK

Faculty of Engineering and Natural Sciences
Master of Science Thesis
December 2019

ABSTRACT

Matti Panula: A Domain Specific Graphical User Interface Framework
Master of Science Thesis
Tampere University
Master's degree Programme in Management and Information Technology
December 2019

Since the early days of software development, there has been an ongoing trend towards higher-order or higher level abstractions in programming languages, software libraries and application frameworks. Some of the arguments for software development tools with higher levels of abstraction include simpler software development, improved portability and better maintainability. Higher level abstractions can however lead to reduced performance. This thesis presents an innovative graphical user interface software solution that mixes high-level and low-level approaches to achieve acceptable performance while retaining good maintainability. The solution is an extension to a graphical application framework called JavaFX.

The scope of this thesis is defined by a software development project which goal is to create a graphical user interface framework. The framework is used in the creation of customer specific user interfaces for an accompanying intralogistics system. The resulting user interfaces must be able to visualize possibly thousands of objects moving on a factory floor. The views must simultaneously support user-initiated zooming, panning, and tilting of the two-dimensional view. Meeting these requirements while maintaining acceptable performance, requires an unconventional solution and a deviation from idiomatic JavaFX.

The user interface framework in question is developed using a high-level graphical user interface application framework called JavaFX. JavaFX is the most recent graphical user interface toolkit included in the official Java Development Kit. It has many reactive traits and other modern high-level properties. Overcoming performance challenges with JavaFX when producing views with thousands of animated items was the key research challenge in this research. Some attention is also given to replacing JavaFX built-in dependency injection system with Spring framework to improve JavaFX suitability to the task at hand.

This thesis presents a hybrid solution that overcomes JavaFX's performance challenges in the problem domain, while retaining as much as possible of the usefulness of the high-level features present in the JavaFX framework. The key innovation is a mechanism that enables automated rendering of sprite-bitmaps from JavaFX scene-graph nodes. The solution includes a system that draws the automatically generated bitmaps to a lower-level JavaFX component called Canvas. The solution enables layered mixing of regular JavaFX views with the custom high-performance views, including seamless resizing and event handling between the two types of views. The solution enables the developers of customer specific user interfaces to choose an appropriate graphics rendering type, such that only objects that cause performance issues, typically items which number exceeds dozens, need to use the more complex high-performance system.

Keywords: GUI framework, JavaFX, Spring, Canvas

TIIVISTELMÄ

Matti Panula: A Domain Specific Graphical User Interface Framework
Diplomityö
Tampereen yliopisto
Johtamisen ja tietotekniikan DI-ohjelma
Joulukuu 2019

Korkeamman abstraktiotason ohjelmointikieliä, koodikirjastoja sekä sovelluskehyksiä on kehitetty kautta tietokoneohjelmoinnin historian. Korkeamman tason abstraktioiden kehittämistä tukee pyrkimys helppokäyttöisempiin ohjelmointityökaluihin. Korkeaan abstraktioasteeseen liittyviä ominaisuuksia ovat myös muun muassa alustariippumaton koodi ja ohjelmistojen parempi ylläpidettävyys. Korkeamman abstraktiotason kääntöpuolena saattaa olla heikko suorituskyky. Tämä diplomityö esittelee innovatiivisen, graafisten käyttöliittymien tuottamiseen kehitetyn, ohjelmistokehyksen joka hyödyntää matalan abstraktiotason ratkaisuja suorituskyvyn parantamiseksi. Kehitetty ohjelmisto on laajennus korkean abstraktiotason sovelluskehykseen nimeltä JavaFX, jota käytetään graafisten käyttöliittymien toteuttamiseen.

Työn rajausta määrittää ohjelmistoprojekti, jonka tavoite on kehittää sovelluskehys jota käytetään graafisten käyttöliittymien luomiseen. Sovelluskehyksen avulla tuotettuja, asiakkaan tarpeisiin räätälöityjä, käyttöliittymiä käytetään sisälogistiikkajärjestelmissä. Käyttöliittymien pitää pystyä visualisoimaan kerrallaan jopa tuhansia sisälogistiikkajärjestelmän kuljettamia tuotteita. Näkymässä pitää pystyä myös navigoimaan sivusuunnissa ja zoomaamaan. Tuhansien liikkuvien tuotteiden visualisointi navigoitavassa näkymässä luo suorituskykyongelman, jonka ratkaisu vaatii tavanomaisesta JavaFX:stä poikkeavan räätälöidyn ratkaisun.

Työn kuvaama sovelluskehys on kehitetty laajentamalla korkean abstraktiotason kehystä nimeltä JavaFX. JavaFX on uusin virallisen Java Development Kit -ympäristön tarjoama käyttöliittymäkirjasto. Se sisältää useita reaktiivista ohjelmointiparadigmaa mukailevia sekä muita korkean abstraktiotason ohjelmointiin liitettyjä ominaisuuksia. JavaFX:n suorituskykyongelmat, jotka johtuvat piirrettävien asioiden suuresta lukumäärästä, määrittävät tämän työn tutkimusongelman. Diplomityö kuvaa myös miten JavaFX:n soveltuvuutta tehtävään parannettiin korvaamalla JavaFX:n riippuvuusinjektiomekanismi (eng. Dependency Injection) Spring-sovelluskehyksellä.

Tämä diplomityö esittelee innovatiivisen ohjelmiston, joka mahdollistaa hyvän suorituskyvyn ongelmakentässä. Ratkaisu perustuu JavaFX:n graafinoodien automaattiseen renderöintiin bittikartoiksi ja piirtämiseen eräänlaiseen hybridinäkymään. Se sisältää myös mekanismin, joka mahdollistaa hybridinäkymän käyttämisen päällekkäin tavallisten JavaFX-näkymien kanssa. Muun muassa näkymän koon muuttaminen ja käyttäjätoiminnot kuten klikkaukset hiirellä toimivat saumattomasti tavallisten JavaFX-näkymien ja hybridinäkymien välillä. Kehitetty sovelluskehys antaa asiakasprojekteja toteuttaville sovelluskehittäjille mahdollisuuden luoda räätälöityjä käyttöliittymiä siten, että ainoastaan asiat jotka saattavat aiheuttaa suorituskykyongelmia piirretään käyttäen uuttaa korkean suorituskyvyn hybridinäkymää ja käyttöliittymän muut osat voidaan toteuttaa käyttäen tavanomaisia JavaFX:n menetelmiä.

Avainsanat: Graafinen käyttöliittymä, JavaFX, Spring, Canvas

PREFACE

When this thesis was started, I was acting as an architect and a lead developer in a software development project that had certain challenges that seemed suitable as a subject for a thesis. I began this thesis as a simple description of a software development project, but sourcing and digesting the existing knowledge turned out to be a much bigger task than anticipated. In addition to a piece of working software and this thesis, the results of this process include a true learning experience in both research methodology and software engineering theory.

I'd like to thank the Finnish education system, the Employment Fund and the faculty for enabling my studies. A special thank you is deserved by my wife Elli who supported me throughout my studies.

Seinäjoki, 16 December 2019

Matti Panula

CONTENTS

LIST OF ABBREVIATIONS

AoP	Aspect Oriented Programming
API	Application Programming Interface
AWT	Abstract Window Toolkit
CPU	Central Processing Unit
CSS	Cascading Style Sheet
DI	Dependency Injection
DOM	Document Object Model
GPU	Graphics Processing Unit
GoF	Gang of Four Design Patterns
GUI	Graphical User Interface
HDL	Hardware Description Language
HMI	Human Machine Interface
HTML	Hyper Text Markup Language
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IoC	Inversion of Control
ISO	International Organization for Standardization
JDK	Java Development Kit
JRE	Java Runtime Environment
MVC	Model View Controller
OCP	Open-Closed Principle
OOP	Object Oriented Programming
POJO	Plain Old Java Object
REPL	Read-Evaluate-Print Loop
RIA	Rich Internet Application
SQL	Structured Query Language
UI	User Interface
URL	Uniform Resource Locator
WCS	Warehouse Control System
WYSIWYG	What You See Is What You Get
XML	Extensible Markup Language

1. INTRODUCTION

This thesis describes methods that augment a graphical user interface (GUI) framework called JavaFX to improve its suitability for a specific use. The work is related to a software development project which goal is to create a domain specific in-house GUI framework and an accompanying reference implementation of the GUI. The resulting user interface (UI) is a cross-platform Java desktop application.

Implementations of the UI are used for monitoring and manipulating an accompanying automation system running on a Linux server. The automation system is typically utilized in warehouse automation. In the scope of this thesis the automation system, acting as a backend for the UI, shall be called a warehouse control system (WCS). The UI shall be referred to as WCS UI or simply the GUI. At the time of writing, the author's employer is rewriting the in-house software platform for creating tailored WCSs. The author of this thesis is in charge of designing and implementing many fundamental aspects of the new GUI for the WCS platform, including the bulk of the solutions described in this thesis.

A key requirement for the new UI is that it should leverage the JavaFX GUI framework. The decision to use JavaFX is likely related to a desire to focus all software development efforts on the Java language. JavaFX was included in Java 8 Runtime Environment (JRE) and Java Development Kit (JDK) released in 2014. It being the most modern GUI framework in the JDK likely influenced the decision to use JavaFX for the new WCS UI. The research goals for this thesis are related to finding, evaluating and implementing ways that make JavaFX more suitable for the task defined in the scope of this thesis.

An international standard, regarding the quality of software systems, highlights 13 quality aspects [25]. This thesis uses two of them, *performance efficiency* and *product maintainability,* as metrics in the analysis of the resulting software artifact. Most of the 13 aspects, the author has little or no control over. Usability, for example, is dictated mostly by a UI specification, though performance does have an effect on usability too. Other things, like security and reliability, are heavily related to interaction with the backend WCS system and are omitted from this thesis. Therefore, the quality of the resulting software is evaluated focusing only on efficiency and maintainability.

Software efficiency is related to resource utilization. Factors such as processing times and throughput are often considered as variables of efficiency. When evaluating software quality from an efficiency viewpoint, things like throughput need to be considered in relation to the target hardware, and also through predefined efficiency requirements.

The maintainability of a piece of software depends heavily on its complexity. Extending, or adding new features, is often required during program lifecycle. If a piece of software is difficult to understand, it is also difficult to extend.

1.1 Efficiency

The frame rate of movies has traditionally been 24 Hz. This means that a new image is projected on a screen twenty-four times every second. To reduce flicker, even in early projectors, the single image was displayed multiple times resulting in typical refresh rates ranging from 48 to 72 Hz. Modern computers usually use a frame rate of 60 Hz, though many computer gamers use much higher frame rates supported by special hardware. Too low frame rates can be perceived as choppy or in other ways unpleasant.

An acceptable frame rate is a key quality metric related to the performance efficiency of the WCS UI. The requirement is not merely visual. JavaFX's default frame rate is 60 Hz. Observed frame rates less than 60 Hz are indicative of a bottleneck, likely a full utilization of a physical Central Processing Unit (CPU) core. This is unacceptable if constant, especially in multitasking environments such as modern desktop operating systems. Logging the frame rate in JavaFX is trivial and can be performed using a class called `AnimationTimer`.

Another efficiency metric in user interfaces is latency. Latency is the time interval between an action and a response. For example, when the user starts panning the view, it should not take too long for the user to see the view move. Many anecdotal sources cite 100 ms as a general limit for immediate UI actions [22], [46]. Much lower latencies are required in applications such as musical instruments or motion control. UI latency can be measured by analysing a video recording depicting the action and the response. The perceived latency challenges during the development of the WCS UI were all observed to be related to issues also causing poor framerate. No further attention on latency is therefore given in this thesis, as focusing on frame rate seems to cover all of the issues.

A customer project that implements the WCS UI may contain hundreds of conveyors that each contain dozens of moving items. This raises some software efficiency considerations. The GUI must be able to visualize possibly thousands of moving artefacts on the factory floor, while simultaneously enabling the panning and zooming of the view in question.

JavaFX employs a reactive programming paradigm and implements a scene graph that resembles the document object model (DOM) familiar from web browsers. Much like the DOM has its limitations, so does JavaFX's scene graph. Overcoming performance efficiency challenges inherent in approaches based on a DOM like scene graph is a top concern in this thesis. The aim is to achieve acceptable performance by augmenting the JavaFX framework using tried and true software engineering approaches.

1.2 Maintainability

Chapter 2 argues that high level code is more maintainable than low-level "bare metal" code. Often the maintainability advantages come at the cost of performance. The performance efficiency aspect in this thesis is related to finding a balance between a low-level programming approach and the benefits provided by the utilization of a high-level GUI framework such as JavaFX. Maintainability in this research is therefore tightly coupled with efficiency. Other maintainability aspects include the software architecture, agile coding practices and code modularity.

There are certain ground rules related to the maintainability of the WCS GUI. For any given customer project, the bulk of the programming and customization happens in the server-side WCS. The GUI should be usable with minimal configuration. In fact, there are two distinct types of projects with different use cases for the GUI framework. Projects that create new UI functionality, and projects that only use the existing functionality provided by the reference implementation of the WCS UI.

A project that requires no new UI functionality should be able to utilize the reference GUI without any customer specific extensions on the client side. This enables project teams to focus on the server side WCS codebase. The GUI framework should therefore allow sufficient run-time customizability. In other words, the reference implementation should be usable in typical small projects without any modifications to the WCS UI codebase.

If a project does require new UI functionality, the new functionality should be as straightforward as possible to implement. A well-structured system for extending the framework also simplifies the creation of new features. If the new UI functionality produced by a

project team is seen as reusable, it can be merged back to the reference implementation codebase. In accordance to Meyer's Open-Closed principle (OCP) [44, p. 54], it is a good practice to implement the new GUI functionality without modifying the existing code in the reference implementation. A structured extension is also easier to merge back to the framework codebase. The merge can be performed at a later time by GUI experts who do not suffer from the pressure of a project deadline.

The maintainability of a piece of software is influenced by software architecture. A clean architecture makes software easier to understand and makes it easier to extend and debug. Some of the conventions in the JavaFX GUI framework were deemed not suitable for the WCS UI. For example, instead of using JavaFX's XML-based views and dependency injection concepts, a custom implementation using Spring Framework was developed. Spring Framework is in a key role in most of the solutions related to maintainability and in making the system run-time customizable and extendable. The use of Spring in augmenting JavaFX is a major factor forming the architecture of the WCS UI and the GUI framework. The software architecture forms a backbone for the maintainability of the GUI.

1.3 Research goals

The goal for this research is to create a domain specific UI framework. The key concept is related to the augmentation of a Java UI framework called JavaFX. This thesis describes the software artefacts that were developed during this research. Full source code is omitted, but the software concepts are presented in a level that enables the reproduction of the results. The software artefacts are evaluated on criterions that are mostly related to performance efficiency and software maintainability.

Chapter 2 will present the scientific background related to the architectural concepts and software tools that are relevant to this research. These include the functional and reactive programming paradigms, and the division of programming concepts to declarative and imperative styles in Section 2.1. Section 2.2 presents common programming concepts with an emphasis to UI programming. The software tools, JavaFX and Spring Framework, are reviewed in Section 2.3.

Chapter 3 describes the study's research methodology. It discusses traditional engineering science and how Design Science and agile software development practices deviate from it. It also highlights the similarities between Design Science and agile software development practices. The evolution of empirical agile practices from Engineering Science

is also discussed. Additional attention is given to software quality as measure used in the analysis of the resulting software artefacts.

Chapter 4 presents the software tools that define the scope for this research. It focuses on the challenges related to using the specific tools in the given problem domain. It discusses both challenges stemming from functional requirements, and challenges caused by the requirement to use a specific software tool, the JavaFX GUI framework. The chapter mostly discusses the functional requirements in the context of JavaFX.

Chapter 5 introduces, in detail, the technical solutions or software artefacts that, along with this paper, form the primary outcome of this research. This Chapter mostly contains descriptions of actual software artefacts. Using the descriptions from Chapter 5, it should be possible for a developer with some familiarity with JavaFX to reproduce similar functionality to that what is described in this thesis.

Chapter 6 analyses the software artefacts that were developed during this research. It focuses on evaluating the performance efficiency and software maintainability of the resulting software artefacts. The final chapter concludes the thesis with a summary of results.

2. BACKGROUND

Software is seldom developed from scratch and GUIs are no exception. Usually a software framework or other types of existing components, such as a software library, provide common functionality to an application. This chapter presents the tools and concepts related to the development of the WCS UI. Section 2.1 about programming languages introduces the concepts and history related to JavaFX's functional and reactive nature. Section 2.2 about design patterns and software architecture presents the relevant concepts and patterns related to UI programming and software maintainability.

GUI software presents special challenges to the developer. According to Myers [45], UIs that are easier to use, are harder to create. Myers also claims that in general UI software is considered to be difficult to debug, modify and implement [45]. There is a trend of web- and other network-backed applications becoming more popular than traditional device specific software [59]. Such network-backed or cloud applications impose additional challenges to UI development. UIs that utilize network services, must be implemented asynchronously and are typically event driven in nature [31]. The reactive programming paradigm is gaining popularity. It is trying to alleviate the problems faced in the development of interactive and event-driven applications [5]. Reactive programming enables the creation of interactive systems without the difficulties present in typical concurrent solutions in imperative languages [52].

The WCS UI is developed using the JavaFX GUI framework. Section 2.3.1 presents JavaFX's core concepts. Spring Framework is an inversion of control (IoC) container that is used in to satisfy the maintainability and configurability requirements in the WCS UI. Spring Framework is presented in Section 2.3.2. JavaFX contains some reactive programming concepts. Reactive programming is discussed in Section 2.1.1. Each of the sections in this chapters are also somehow related to software maintainability. Section 2.1.2 about functional programming describes how functional programming concepts can improve software maintainability.

Due to their pervasive nature in the field, basic descriptions of procedural and object-oriented programming (OOP) concepts are omitted. Readers interested in procedural programming are referred to Kernighan and Ritchie's classic book on the C programming language [32]. To learn advanced object oriented design, one could consider reading

Design Patterns [19]. The book is often called Gang of Four (GoF) after its four authors. Some of the GoF book's patterns are presented in Section 2.2.

2.1 Programming language paradigms

This section presents the concepts of imperative and declarative programming and provides a glimpse to the history leading to JavaFX and reactive programming. Early programming languages were modelled after the underlying hardware they were run on. As more computers were developed, it became necessary to develop a language that was portable. Portable languages could be compiled to run on different types of computers. The story of higher level languages began in the 1950 with Fortran that gained popularity not least due to its portability [24]. Another milestone in higher level language development is ALGOL which introduces *Structured Programming* concepts that are still widely used today. These include code blocks, loop statements and conditionals [65]. The term "Structured programming" was coined by Edsger W. Dijkstra. Dijkstra also wrote the famous open letter titled "Go To Statement Considered Harmful" promoting the structured approach to programming [12].

Programming languages can be classified into families based on their model of computation. Scott divides programming languages into two top-level classes, imperative and declarative [55]. Declarative languages focus on what the computer should do, whereas imperative languages focus on how the computer should do it. It has been argued that algorithms can generally be described as consisting of logic and control [34]. Declarative programming is focused on presenting the logic of the computation [38]. Imperative programming on the other hand focuses on manipulating state using statements that form the control flow [24]. JavaFX is related to reactive and functional programming styles, both of which are declarative approaches. These are discussed in more detail in sections 2.1.1 and 2.1.2.

The most widely used declarative language today is arguably Structured Query Language (SQL) that is used to interface relational databases. When issuing a declarative SQL query, the database management system typically uses a query optimizer that generates an imperative query plan that is used to access the data. Another example of language that supports declarative programming is Lisp. Lisp is a language invented in the late 1950s. After Fortran, it is the second oldest language still in widespread use [1],[60]. A more tangible examples of declarative programming are perhaps hardware description languages (HDL). HDLs emphasize logic over control flow [21]. Many HDLs

can be synthesized to form a physical implementation with programmable logic gates [33]. In such cases, the HDL defines the logic formed by the physical gate network. The circuit then continuously reacts to external input. In practice constructs like a main-loop or a state machine are often manually added to HDL designs for handling the control flow.

Imperative languages are based on John von Neumann's model of machine execution [27]. Fortran and ALGOL are both imperative languages. C is an example of a basic imperative language that is widely used today. Imperative languages remain more popular than declarative, not least due to performance reasons [55].

Object orientation is probably the most common programming language paradigm in use. According to TIOBE's programming language popularity ranking, Java and C++ have been part of the top three of most popular languages since the turn of the millennium [60]. Despite their structured and distributed model of memory and computation, object oriented languages such as Java, C++, C# and Python, are classified as imperative languages [55]. The first object orientated language was SIMULA 67 [55]. It was developed in the University of Oslo by Ole-Johan Dahl and Kristen Nygaard and published in 1967. The developers of Smalltalk from 1970s and C++ from 1980, which are more recent and popular object oriented languages, both cite SIMULA as a major influence [10].

Java 8, published in 2014, introduced declarative concepts such as lambda-expressions and streams that are generally labelled under the functional programming paradigm. Due to challenges posed by distributed and parallel computing among other things, declarative and functional programming concepts are gaining popularity in both server [63] and GUI programming. Microsoft's C# introduced lambda expression in 2007 [61], and Apple's Swift from 2014 contains many functional aspects such as the read-evaluate-print-loop (REPL) [67]. JavaScript, the language of the web, is also related to Scheme which is a dialect of the functional language Lisp [53].

2.1.1 Reactive programming

JavaFX GUI framework employs a reactive programming paradigm. JavaFX is included in the standard JDK since Java 8. Reactive programming is closely related to the concepts of dataflow programming. Various applications, such as music production software, video games and web-pages employ an embedded dataflow-like language [7]. Reactive and dataflow programming are, like functional programming, classified as members of the declarative programming paradigms.

Much like electronic circuits, reactive programs operate with continuous values that vary over time. Reactive programming is focused on the propagation of change [5]. Reactive programming simplifies the creation and maintainability of declarative event-driven software. It allows the programmer to express what the UI should do and let the reactive extension or language runtime mange the implementation.

The reactive JavaFX example below creates a text-field that displays a sum of two integer properties. As long as the text-field is visible, whenever either of the integer properties value changes, the summation and string conversion is recomputed and the resulting string is displayed by the text-field:

```
TextField txt = new TextField();
IntegerProperty int1 = new SimpleIntegerProperty(2);
IntegerProperty int2 = new SimpleIntegerProperty(2);
txt.textProperty().bind(int1.add(int2).asString());
```

The API document on JavaFX bindings [49] states that bindings are calculated lazily. When dependencies change, the binding's result is not recalculated, but it is marked as invalid. When the invalid value of a binding is requested, only then is the value recalculated. If the text-field in the example above would not be visible the binding would remain invalidated until its value is requested.

Bainomugisha describes the reactive paradigm as being spreadsheet-like [5]. To emphasize the similarity with spreadsheets, the two integer properties in the example above could be exchanged for `TextField` objects. The three text-fields would then form a kind of three cell spreadsheet with the summation function applied.

To replace the summation with subtraction, for example the following is possible:

```
NumberBinding subtr = Bindings.subtract(int1, int2);
lbl.textProperty().bind(subtr.asString());
```

Implementing a spreadsheet-like application with JavaFX's bindable properties would be quite straightforward compared to many traditional approaches involving callbacks or the observer-pattern. The more traditional approaches are discussed in Section 2.2.3.

The dependencies in a reactive program form a directed graph called a dependency graph. The propagation of change throughout the dependency graph is called data flow. Reactive programming utilizes the synchronous dataflow programming paradigm [37] without the strict real-time requirements. Continuous time-varying values are represented as behaviours and discrete values by events. Unlike in synchronous dataflow, the reactive paradigm also allows for the structure of the dataflow to change over time [9], [56]. Much of the research on reactive programming is based on Fran, a functional

reactive animation extension to the Haskell programming language [5]. Hudak and Elliot published their paper on Fran [15] in the year 1997.

The reactive terminology is not limited to GUI programming. A recent and general description of reactive systems is provided in The Reactive Manifesto [6] which discusses reactive systems in a broader sense for example in the context of distributed server systems and distributed applications.

2.1.2 Functional programming

Imperative languages have implicit state that is modified by commands such as variable assignment or the while loop. Functional languages in contrast have no implicit state and the computation is carried out solely through the evaluation of expressions [24]. Functional languages are considered more *"high-level"*, which refers to the origins in mathematical notation.

The mathematical style entails the notion of immutability, or the lack of side effects. Pure functions must always return the same results with the same parameters. The lack of side effects can be seen as a discipline for good programming [24] much like the avoidance of goto-statements [12]. However, like the spirit of the goto-statement still lives in the break-statement, even pure-functional languages must deal with some side effects and state. Input and output, for example, require both state and side-effects. The functional approach to dealing with state is explicit rather than implicit like in imperative languages [24].

Functional programs' syntax can be very similar to mathematical notation. For example, the factorial of a positive integer n can be presented in recursive mathematical notation such that $n!$ = fact(n) where:

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ \text{fact}(n-1) * n & \text{if } n > 0 \end{cases}$$

The same recursive approach can be implemented in Java 8 using a functional programming style. The function is marked final to reflect the important functional concept of immutability:

```
final Function<Integer, Integer> fact = n -> {
    if (n==0) return 1;
    else return this.fact.apply(n-1)*n;
};
```

The previous code example could have been written as a regular method. However, the example presents, in quite an unpractical way, the Java 8 syntax for both the lambda-expression and a first-class function. Java 8 introduced higher-order functions [62]. Such functions can take functions as parameters and return a function as a result. A function that can be assigned to a variable is called a first-class function. In addition to lambda expressions, like in the previous example, regular methods can also be used as first class functions in Java 8.

A more graceful way for calculating factorials using functional Java would be the following, where the `rangeClosed` -method creates a stream, or a sequence, of numbers (1…n). The stream is consumed by the reduce function:

```
IntStream.rangeClosed(1, n).reduce(1, (int a, int b) -> a * b);
```

This approach is identical with the more common mathematical definition of a factorial:

$n! = 1 * 2 * 3 * \ldots (n-2) * (n-1) * n$

The benefit of reduce, compared to mutating a running product in a loop, is the more graceful parallelization without the need of additional synchronization [49]. Both the stream and reduce are traditional functional programming concepts. Streams can be used to generate infinite sequences, for example the set of all positive integers can be generated with Java 8's `IntStream` as follows:

```
IntStream.iterate(0, i -> i + 1);
```

Obviously generating an infinite sequence is impossible. The above example is also limited by the size of a 32bit integer. The items in the stream are actually calculated only when they are needed. Such deferred computation style is called lazy or non-strict evaluation. A similar concept is present also in JavaFX's lazy evaluation of bound values, and in Spring-framework where it is possible to annotate beans with `@Lazy` to defer initialization.

Distributed software systems and efficient utilization of modern multi-core central processing units (CPU) require concurrent computing. Concurrent programming is challenging. Race conditions, deadlocks and concurrent update problems are caused by mutable variables. As variables in functional languages are immutable, functional programming is a good solution to many concurrent problems [39, p. 70]. Functional programs also have less overhead or "boiler plate"-code than their object-oriented counterparts. Such shorter programs are cheaper to maintain, build and deploy [43].

2.2 Design and architectural patterns

This section presents the relevant design patterns and architectural concepts in JavaFX and the WCS UI. Architecture is a term that many people associate with plans used in building houses or machinery. Software architecture can be analogously thought of a plan that is used in software development. Even though it is possible to write a piece of software in a single file without functions or classes, such a program would arguably be difficult to maintain. Unlike the building blocks of houses, the components defined by software architecture are abstract, immaterial and often difficult to conceptualize. Martin describes software architecture as a tool for planning appropriate boundaries between software components [39]. The components defined by software architecture may reflect the boundaries between real-world objects but they can also be much more abstract.

Design patterns is a term that was popularized in computer science context by the popular GoF-book [19]. The book presents 23 object-oriented design patterns in C++ and Smalltalk. One of them, the observer is discussed in Section 2.2.3 as an essential built-in feature of JavaFX. There exists a vast number of anecdotal commentaries for and against the usage of design patterns in the web. Zhang and Budgen argue that there's no proof on the effectiveness of OOP design patterns from the GoF book, but they do suggest that the patterns are sometimes useful in improving software maintainability when the use of patterns is documented appropriately [66]. Norvig suggests that the patterns in the GoF-book are mostly limitations in C++ and Smalltalk and shows that most of the patterns can be greatly simplified by using a more capable language like Lisp [47].

Recurring design does exist in non-OOP contexts. One could conclude that concepts like *data hiding* via *encapsulation* or *inheritance* are design patterns. However, the term "design pattern" seems to generally imply an OOP context. Both the GoF-book and most of Martin's work are related to OOP. The patterns in software design are empirically derived generalizations and seldom without drawbacks. Choosing the most suitable pattern or abstraction is often non-trivial. Data hiding, for example, generally makes the code more maintainable, but global state is still often favored due to other aspects or constraints such as performance efficiency.

2.2.1 Dependency Injection

Martin proposes that to achieve a clean object-oriented architecture, classes should strive to depend solely on abstract declarations [39, p. 100]. In Java this means that only

abstract classes or interfaces should be used as dependencies in the source code. This is of course impractical as one must typically depend on concrete implementations of classes such as the Java `String`. A developer can however rely that the `String` class, or more specifically the interface and visible behaviour of it, will not change. Martin calls such classes stable. Stable classes can be referred to directly, but volatile classes should be abstracted behind stable interfaces.

When class A depends on interface B and class C implements B, the dependency is inverted. Both A and C have a dependency on B but not each other. Dependency Injection (DI) and Inversion of Control (IoC) are techniques used in OOP, that enable the injection of object dependencies via an external service. According to Fowler [16], the term DI was initially used in a 1988 article titled “Designing reusable Classes” [28]. Martin states that the concept was already present Xerox’s internal functional specifications in 1980 named as “Don‘t call us, we’ll call you (Hollywood’s Law)” [58]. Later literature often refers to it as the “Hollywood principle”. Martin’s 1996 article introduces a related concept as “The Dependency Inversion Principle” [40].

Java Platform, Enterprise Edition (Java EE) specification defines “lifecycle contexts” and “dependency injection” among other things. DI frameworks, such as Spring, are abundant in the Java ecosystem. DI-frameworks simplify the handling of the object instantiation and lifecycle. Similar concepts can be found in UI development. The views in JavaFX can be defined in code, or by using an XML-based notation FXML [68]. Elements defined in the FXML-file can be accessed in Java code by marking an undefined variable with a specific annotation. The JavaFX context will then inject the dependencies defined in the FXML to the annotated variables.

2.2.2 Model View Controller

Model View Controller (MVC) is a well-known GUI programming phrase, but the MVC architecture may be less known. Like many other GUI programming concepts, it has its roots in Smalltalk, one of the first OOP-languages. Fowler highlights two aspects of MVC that are still relevant today, Separated Presentation and Observer Synchronization [17]. Separated presentation is the emphasis on the separation of data or model, and the presentation which includes the view and controller components. Observer Synchronization is related to the way a presentation layer reacts to the changes in a model.

Reenskaug, the original developer of MVC, describes MVC as mental model or a Pattern Language that helps developers to discuss about the business domain in terms of objects [54]. This is related to the GoF-book's behavioural patterns that *"… shift your focus away from flow of control to let you concentrate just on the way objects are interconnected* [19, p. 245]*".* Components that handle user input, such as buttons or text boxes, are called controllers in JavaFX. JavaFX also inherits much from MVC, for example, in the way objects in the presentation layer can be wired to observe values in the model.

2.2.3 Observer

The Observer is a behavioural design pattern presented in the GoF-book. Behavioural patterns model the run-time behaviour of objects. Java has a build-in support for Observables and JavaFX extends the GUI centric Observer-functionality even further [49]. For example, a spreadsheet can present the same data with different kinds of charts. The charts may be independent of each other but have subscribed to observe the same data. The Observer pattern implements the publish-subscribe model with plain objects [19]. JavaFX has a vast array of tools that implement concepts that are based on MVC and the observer pattern.

2.2.4 Actor

The Actor Model is a mathematical model for concurrent computation, that is characterized by concurrency, resilience and fault tolerance [4]. This section presents the Actor Model as an alternative point of view to help the reader evaluate the limitations of MVC and Observer based approaches such as JavaFX. JavaFX is heavily influenced by Reactive concepts, but as the presentation always happens in a single thread, it cannot be considered as a pure concurrent reactive framework.

Callbacks are a concept that aim to solve the same problems as the Observer pattern. They offer a solution to the problem when, for example, a view has two values that are dependent on each other. In such cases it is possible to defer the checking of the constraint using a callback. JavaFX's lazy evaluation of bound observables can be seen as a related concept using a simpler syntax, but with a limited functionality as a drawback. Heavy use of callbacks can lead to a situation called *Callback Hell* [14]. The Actor Model, which is closely related to Reactive and Dataflow programming, offers a solution for such problems. Commonly used Actor Model implementations in the Java ecosystem are the Akka-framework and the Scala programming language. JavaFX's thread-model makes

callback handling simpler, but many practical JavaFX applications still use callbacks and require writing additional multi-threaded code that may cause maintainability challenges.

2.3 Frameworks and libraries

This section presents the two main pre-existing components or frameworks in the WCS UI, JavaFX and Spring Framework. The GoF uses the term *"inversion of control"* (IoC) when describing software frameworks as a defining factor for software architecture [19, pp. 26–27]. When a framework is employed, the flow of control is inverted. The developer is writing code that is called by the framework. JavaFX applications, for example, are started by instantiating an implementation of an abstract class called `Application` [69]. Similarly the Spring Framework is launched by instantiating an implementation of an interface called `ApplicationContext` [70]. In both JavaFX and Spring, it is the framework that calls code written by the developer.

GUIs are seldom developed without utilizing existing functionality such as a GUI framework [45]. Despite the ubiquity of high-performance graphics processing units (GPU) in modern hardware, many games are still developed in low-level, imperative languages with manual memory management. Most GUIs, GUI-frameworks and web browsers currently utilize hardware accelerated graphics. Regardless of the available GPU resources, one should use consideration when choosing a convention, style or approach that best suits the task at hand. Creating forms with a game engine or developing games with a form-oriented widget toolkit can be possible, but it is likely impractical. GUI frameworks, and software frameworks in general, impose conventions and canonical ways for doing things. When high performance efficiency is a key requirement, high level abstractions are typically less suitable for the task.

Rich Internet Applications (RIA) developed using Adobe Flash, Microsoft Silverlight or Java's "Applets" web browser plugins can be used for embedding content into webpages. Before web browsers became more capable, RIAs were the preferred solution for many things such as a business-oriented Java Applet GUIs or games based on Flash.

2.3.1 JavaFX

JavaFX was introduced by Sun Microsystems in 2008. It was initially targeted for the creation of RIAs and mobile applications [48]. Early JavaFX applications were written using a declarative scripting language called JavaFX Script instead of the Java language. JavaFX Script could be compiled to Java bytecode and run as a Java Applet in the

browser. Using a new scripting language instead of Java was likely intended to attract content creators familiar with JavaScript and Flash. JavaFX did not become a popular mobile framework or a serious competitor to Flash. In 2011 Oracle, who had acquired Sun, introduced JavaFX 2.0 that dropped JavaFX Mobile and enabled the creation of JavaFX applications using the Java language. With the introduction of JavaFX 2.0, Oracle also announced its intentions on open-sourcing JavaFX. The future of JavaFX seems to be tied to OpenJDK as Oracle has hinted that JDK 11 will no longer ship with JavaFX. At the time of writing, JavaFX seems to be mostly used for creating business applications for the desktop. Considering its impact on the industry, JavaFX can be seen merely as a modernized alternative to older Java graphics toolkits such as Swing and Abstract Windows Toolkit (AWT). Some of the features that distinguish JavaFX from its predecessors include the declarative programming approach, similarities with web-development and improved touch-screen support.

One of JavaFX's basic concepts is the scene graph which resembles the Document Object Model (DOM) standard used in web browsers. JavaFX provides common components, such as text input, buttons and tables, that are present in web browsers and typical desktop widget toolkits. Much like most web browsers, JavaFX has an element called `Canvas` that can be used for developing more efficient graphics. Such feature is required to enable the implementation of many types of games for example. The `Canvas` provides low level abstractions that enable direct manipulation of pixels, and efficient drawing utilities for two-dimensional graphics including bitmaps, text and polygons.

2.3.2 Spring framework

Spring is an open source IoC framework for the Java ecosystem. It is based on 30,000 lines of sample code from Rod Johnson's book *"J2EE Design and Development"* [64]. The book presents good practices and example implementations for the predecessor of JavaEE, the Java 2 Enterprise Edition (J2EE) [29]. Spring was released less than a year after the book as an open source project led by Juergen Hoeller and Yann Caroff [30]. Spring is a serious alternative for the Java Platform Enterprise Edition (JavaEE) standard [2]. Google trends indicates that Spring is more popular than JavaEE [20].

Today Spring is a broad collection of different extensions called *projects*, but the Spring Framework remains in the core. It emphasizes the concept of "convention over configuration" and claims to be an "opinionated framework". This means for example, that creating a database backed web application can be done with a few lines of code. Extending

and overriding the default configurations enable the creation of more complex custom systems. Spring's project description defines Spring framework as a "programming and configuration model for Java-based enterprise applications" [51]. Spring framework is based on the concept of *"coding to interfaces"* [64]. This enables a loose coupling between the caller and the implementation. Spring's application context handles the instantiation of classes that implement the interfaces. Spring is an IoC container where the context handles dependency injection (DI) and aspect oriented programming (AoP) as the core functionalities [30]. AoP enables instrumenting methods with, for example, logging or testing related code without adding code to the instrumented class. This provides means for defining component boundaries based on aspects such as logging or testing.

2.4 Summary

This chapter presented the history and the evolution leading to tools such as JavaFX and Spring Framework. Dependency injection was presented as a part of JavaFX. It was suggested that as an independent component in JavaFX, the DI functionality can be replaced with other DI frameworks such as the Spring Framework.

A gradual paradigm shift from imperative languages towards declarative languages was argued to be in progress. The shift seems to correlate with constantly increasing computing power. Both JavaFX and Spring Framework were examined from a declarative and functional programming viewpoint. It was noted that many functional concepts such as streams or lazy initialization are common in many of the components in contemporary Java ecosystem. Some attention was given to the increased application of reactive programming and the relation of the reactive approach to the pervasiveness of distributed computing.

This chapter presented the balancing act between the declarative and imperative approaches, and the challenges related to producing maintainable code. This, in the WCS UI problem domain, forms the research question for this thesis. The declarative JavaFX's performance challenges in this domain may be related to the intended use of JavaFX being forms, and other simpler UIs. The lack of sufficient computing power in modern computers is likely an insignificant factor. Falling back to a more imperative style of programming using a `Canvas` was hinted as a solution to the performance challenges faced with idiomatic JavaFX. This will be discussed with more detail in later chapters.

3. RESEARCH METHODOLOGY

This chapter presents Design Science as the primary research methodology employed in this research. Other concepts related to research in both engineering and software development are also presented in this chapter. Some practical aspects related to employing design in software development contexts, such as Agile and Lean, are touched in Section 3.3.

The goal of this research is to generate a solution concept, implement it in software, and evaluate the implementation of the new software artefact. The software in question is part of the WCS UI framework called factory-view. As the study is related to creating a man-made artefact, a natural science approach to research could be considered sub-optimal. Natural Science is mentioned here as an example of traditional science, in contrast to Design Science.

3.1 Engineering Design

Software development is related to engineering, but also design. Asimow portrays the engineering design process as a sequential feedback loop [3]. Each design operation's outcome is evaluated, and based on the results of the evaluation, the operation is either repeated or the process continues to the next step. New information is acquired on each evaluation round. It is essentially a knowledge gathering process.

Dieter and Schmidt use the terms synthesis and analysis to describe the implementation and evaluation phases in engineering. They also emphasize the process of decomposing complex problems to more manageable parts as part of the analysis [11, p. 2]. The software development process used during this research used a similar iterative model. Each iteration or version of the software created also new information and insight regarding the inner workings of the JavaFX framework and the way it could be utilized in the problem domain.

Natural sciences are focused on natural things. Engineering, as well as any other profession, is focused on the artificial. Work that aims to change an existing situation in to a more preferred one is commonplace. Such work happens when doctors prescribe medication, or when new policies are devised by the state. According to Herbert Simon, all such activities can be thought of as design. Simon defines design as "… the principal

mark that distinguishes the professions from the sciences.” [57, p. 111]. Dresch et al. propose that Simon “inspired the distinction between exploratory sciences (traditional science) and the sciences of the artificial - Design Science “ [13]. Software development patterns regarding zoomable high performance views using JavaFX do not exists. Therefore, instead of purely using traditional engineering methodologies, the process of design needed to be employed in the development of the novel solution used in the factory-view component.

3.2 Design Science

Dresch et al. [13] describe Design Science as “… a form of scientific knowledge production that involves the development of innovative constructions …” with the intention to solve real world problems while producing a scientific contribution. The outcome can be “… an artefact that solves a domain problem, also known as solution concept, which must be assessed against criteria of value or utility”. Chapter 4 presents the criteria for the artefact produced during this research. Chapter 5 presents the actual artefact and Chapter 6 assesses the artefact in relation to the criteria presented in Chapter 4. The criteria can be summarized as forming of two quality metrics, efficiency and maintainability. Both of these are however entwined with domain requirements and cannot be assessed independently.

Software efficiency is often quantizable. Metrics such as the number of instructions per second, or the memory consumed on the target hardware are generally trivially measurable. The maintainability of a software system is however much less tangible. Both goals can however be promoted by utilizing the process of *design*. Design is also vital to the process of creating new artefacts, such as software. Much of the value and utility provided by the solution concept are related to technical aspects. The software development process is however a vital part of the design process. When software is being developed, knowledge about software design is generated simultaneously.

The process in design science can be seen as consisting of three parts, the relevance cycle that relates to defining requirements and acceptance criteria, the rigor cycle that is related to past knowledge, and finally the central design cycle that is related to building and evaluating the innovative artefacts [23]. Figure 1 demonstrates the three cycles.

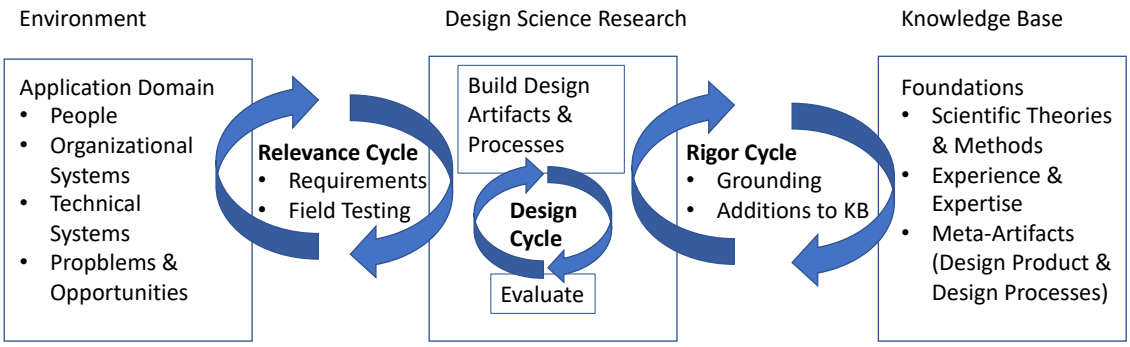

Figure 1. *Design Science Research Cycles, adapted from [23]*

3.3 Design in Software Development

The iterative process of synthesis and analysis in engineering design resembles Fowlers characterization that typical software development uses a “code and fix” approach. The difference is that traditional engineering design is based on standardised components and accurate calculations. Fowler states that the “chaotic” “code and fix” software development process only works for small projects [18]. He continues to describe that traditional engineering methodologies were initially used to try to make software development more predictable and efficient.

The use of traditional engineering methodologies with software development is however often critiqued as being bureaucratic. Agile methodologies were developed as a way to add just the right amount of bureaucracy or “process” to gain a reasonable payoff [18]. There are similarities between Agile and Design Science approaches, and studies that aim to unify the Agile and Design Science approaches as a scientific method exist [8], [50].

The WCS UI software has been developed utilizing an adaptation of an Agile software development framework called Scrum. Agile is an umbrella term that covers a number of lightweight software development methods including Scrum and Extreme Programming. The Manifesto for Agile Software Development, signed in 2001 by many influential industry professionals, is often mentioned as a key milestone in the history of Agile software Development [41]. Scrum allowed the inclusion of design as part of the software development process. This led to an iterative development process that, in the spirit of Design Science, creates new knowledge on each iteration.

Agile shares with software quality research, the sentiment that people are the biggest influence in the quality of software systems [36]. Design Science and Engineering Design

on the other hand focus less on people and interactions. Design Science can be thought of as an attempt in gaining scientific knowledge from studying crafts such as engineering. The agile movement, on the other hand, often advocates software craftmanship. Software craftmanship extends the concepts from Agile by for example, emphasizing developer professionalism, accountability and the growth from an apprentice to a master.

Agile software development shares alikeness with Design Science and Lean manufacturing. Lean focuses on reducing waste and gathering information in the entire supply chain. In agile software development, knowledge can be gathered by, for example delivering the customer a very early version of the software. The customer can then provide early feedback. This reduces waste as early feedback makes software design more efficient. In this project, the performance limits of the JavaFX framework were communicated to stakeholders in an early stage of the project. This affected the scope of the project and led to the development of the custom, high performance solution described in this thesis.

3.4 The research process used in this thesis

This thesis presents the results of a research that is a part of an agile software development project in the field of human machine interface (HMI) software development. The principles of design science constraint the scope of this thesis. As such, attention is only given to parts of the software development project that generate JavaFX HMI related knowledge that can be useful to other professionals in the field.

The research began with a design science concept that Hevner calls the relevance cycle [23] in which the requirements and acceptance criteria were defined. These quality related aspects are discussed thoroughly in Section 4.1. Once the requirements were defined, the research proceeded to the design cycle [23] that begun with the study of the features and limitations of the JavaFX framework.

In the design cycle, many unsuccessful implementations were developed and with each iteration the performance of the developed artefact was evaluated. Each iteration revealed limitations, but also gained insight on the inner workings of JavaFX. This process paved the way for the key innovation presented in this thesis. Once the pros and cons of both the reactive node-based approach and the low-level `Canvas` in JavaFX had been sufficiently studied, the resulting hybrid solution was discovered.

The rigor cycle [23] in design science is related to making use of existing knowledge. The new innovative artefact is also evaluated in the light of the existing knowledge. Discovery of the concept that high-level nodes could be used to generate bitmaps likely resulted from studying JavaFX but also from earlier graphics software development experience with environments using for example, low-level C or high-level JavaScript. Such use of existing knowledge is key in design science [23]. Existing solutions were researched, and many related concepts and implementations are presented in this thesis [26], [42], [35]. Building on existing research is crucial in the sense of maintaining high scientific standards and rigor in design science research.

4. REQUIREMENTS AND PRECONDITIONS

This thesis is focused on a specific part of the WCS UI, namely the *factory-view*. The aim is to implement a solution that provides both sufficient performance and maintainability while utilizing the JavaFX framework. This chapter introduces the factory-view and the technical aspects related to implementing it. Overcoming performance challenges without deviating too far from idiomatic JavaFX is the top concerns of the research.

4.1 Quality requirements

This research evaluates the factory-view implementation from two viewpoints, performance efficiency and software maintainability as defined by ISO/IEC 25010 [25]. The quality of a piece of software is mostly related to its fitness for the purpose it was developed for. Therefore, the fitness to purpose cannot be defined unless the purpose has been appropriately specified.

The efficiency requirement in this research is related to achieving a sufficient screen refresh rate. In JavaFX this is typically 60 Hz as it is JavaFX's default refresh rate. Idiomatic JavaFX with separated presentation sets a good benchmark for the maintainability. Later sections of this thesis show that performance reasons dictate a deviation from idiomatic JavaFX and that the balancing act between maintainability and performance can be far from trivial.

Screen refresh rates below 60 Hz may look unpleasant and in JavaFX are indicative of a performance problem. It is a safe guess to say that using JavaFX's `Canvas` would likely provide good enough performance. The `Canvas` provides high-performance low-level drawing primitives. Such an approach, while performant, would however likely lead to poor maintainability. The sub-par maintainability of close-to-the-metal graphics frameworks is surely one of the reasons why higher-level UI frameworks and toolkits, such as JavaFX, exist.

The level of maintainability required is not easy to define. As using JavaFX was a predefined requirement, it is safe to assume that idiomatic JavaFX provides a sufficient level of maintainability, and as such a level of maintainability to aim for. Using JavaFX Nodes

in separate views with injected controllers provides a clean architecture that should provide good enough maintainability. New custom solutions should not increase the complexity experienced by a developer using the WCS UI framework in customer projects.

The quality viewpoints from ISO/IEC 25010 [25] that are omitted from this thesis include any metrics regarding reliability, security or code quality for example. The software development process, described in Section 3.3, about agile software development processes, is however also related to software quality as defined by ISO/IEC 25010 [25].

4.2 The factory-view

This section introduces the factory-view and presents many of the requirements related to the drawing of the items visualized by it. A high-level description of some of the solutions are also provided in this section. The factory-view is the most important part of the UI and also the most complex. The factory-view is a two-dimensional blueprint-like top-down view of the actual factory or warehouse. It is a zoomable view that contains several layers of information. Many of the layers can be either hidden or visible at a certain time depending on the zoom level and other factors.

One of the most fundamental tasks of the factory-view is to visualize the material flow in the conveyors of a factory or an automated storage. An implementation of the factory-view may have to visualize dozens of conveyors spanning several kilometers. The conveyors combined may carry thousands of transportation units (TU) that must occasionally be visualized simultaneously.

Figure 2 illustrates a simplified conveyor carrying two pallets. The leftmost pallet in Figure 2 contains 4 stacks and each of the stacks contains two boxes. The number of boxes i.e. stack height is indicated with a label containing a number (the actual height of the stack is not relevant in the scope of drawing the factory-view).

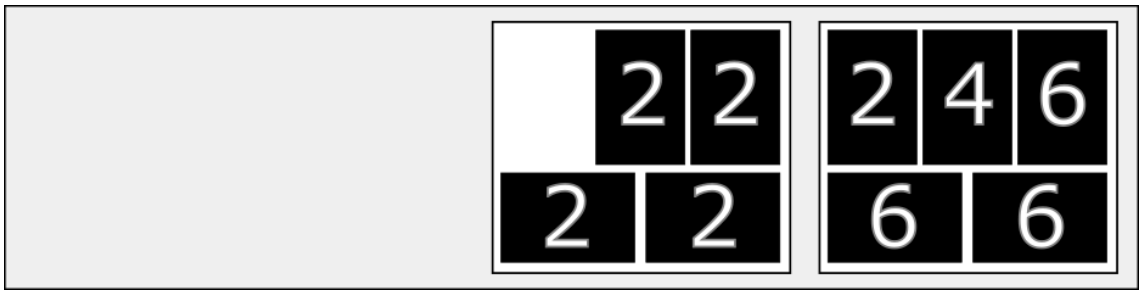

Figure 2. *Simplified illustration of a conveyor transporting two pallets*

In an actual implementation, several interconnected conveyors in different orientations are present, and the view is typically decorated with status colors, direction arrows and other helpful visual cues for the factory operators. Still, even the example in Figure 2 can be extrapolated to reveal several design challenges. The illustration presents 4 layers of information, the conveyor device, the pallet, the stacks and the labels that indicate the number of boxes in the stack. Visualizing such tree-like information, while simultaneously maintaining layer abstractions, requires a separate mechanism that keeps track of the layer or z-axis depth of a node.

In addition to simple conveyors, a typical factory floor can contain intersections, stackers, gantry robot storages and many other things. The details of the various devices will be omitted for brevity. It should be noted however that, even though the conveyors are used as an example, the requirement to support various different devices with TUs drawn over them in varying patterns is a requirement affecting the design.

Additional requirements include the possibility to rotate the contents of the factory-view in 90-degree increments to reflect the orientation of the desktop computer in the factory floor. The orientation of the labels must however always remain horizontal and remain positioned to the centre of the labeled item. The labels that indicate the stack heights are only drawn at appropriate zoom levels. If the user zooms closer, even more information is drawn over the TUs. All of the text-labels are omitted entirely if the zoom-level would make them too small to be readable. Such requirements present some implementation challenges, especially regarding performance efficiency. Efficient visualization of the thousands of TUs along with the maintainability aspects provide the scope for this research.

4.3 Scaling and translation

Scaling and translation are geometric transformations. Scaling is visually similar to zooming with a camera, and translation has similarities with panning and tilting. This section presents the basic concepts of such calculations as implemented in the factory-view.

Scaling and translation do not involve the perspective changes that are present when zooming and panning using a real camera. Though not entirely accurate without explicitly specifying an orthographic projection, this thesis uses the terms zoom, tilt and pan to describe what the user of the UI is either seeing or doing.

The WCS UI may visualize factories spanning several kilometres. The user of the UI may at times wish to view the entire factory, for example to easily spot a device in an error state. The user may also wish to view the contents of a single pallet on a conveyor. To facilitate the visualization of items with vast differences in sizes, the factory-view can be zoomed in and out.

A single decimal variable can be used to represent the zoom-value, or scale. The coordinates and dimensions of all drawables can simply be multiplied with the scale-variable. This is enough to make the view scalable, but such scaling happens around the origin point. When zooming in using a scroll-wheel on a mouse, or a zoom-gesture on a touch screen, it is common that the mouse pointer or the central point of the zoom gesture defines a kind of a pivot point for the zoom action. This makes it possible to directly zoom in to an area of interest without the need of a separate panning operation.

It is possible implement the zooming and the panning in such a way that both the pivot of the zoom, and the translation operate using the same two translation variables, we shall call them x and y offset. When the user pans the view to the left by dragging with a mouse, the x-offset variable can simply be subtracted by the number of pixels the mouse pointer is traveling. In JavaFX, the x-offset can be a property bound to the scaled x-coordinate property. This would enable visualizing the panning operation while it is happening and would not require any additional draw operations. Such thing is relatively simple to implement in JavaFX when using nodes, as the framebuffer is refreshed automatically and the programmer needs not to worry about, for example, manually calling draw after every operation.

When the user zooms to an area of interest, it is required that the entire selected area remains visible even when factory-view component gets resized. The resize is a common occurrence, that happens when another component of the WCS UI grows, and steals

screen real estate from the factory-view. To facilitate keeping the selected area visible, the scale and translate operate on a rectangle instead of just a single point describing the x and y offsets. The selected-area-rectangle is changed whenever the user zooms or translates. The pivot point for the scaling is the in middle point of the rectangle. Zooming in shrinks the selected-area-rectangle and zooming out grows it. The draw calculations are then implemented so that the real-world area defined by the selected-area-rectangle is always zoomed in the centre of the factory-view. The scale is then defined by either the horizontal or the vertical dimension of the rectangle, whichever is the limiting dimension, depending also on the aspect ratio of the factory-view.

4.4 JavaFX Scene Graph

JavaFX's Node-based scene graph is well suited for many tasks. A graph-based approach feels natural for developing forms and documents that may contain images, text input, buttons and sub-forms for example. The scene graph is very similar with the Document Object Model (DOM), a tree-like data structure used in web browsers to represent HTML nodes.

JavaFX contains many different layout container classes. Their super class is called a Pane. Pane is subclassed by classes such as GridPane and VBox for instance. As the names might suggest, the GridPane helps in positioning nodes to a grid and the VBox allows positioning nodes next to each other vertically. Such layout containers can be very useful in typical form-like applications. However a top-down view similar to a blueprint, like the factory-view, can not utilize such automatic layout. The drawables of the factory-view must be drawn on specific coordinates that depend on the current panning and zooming, and also map to real-world coordinates of the object that the drawable items represent.

To position nodes to specific coordinates, one can use JavaFX's top-level layout container called Pane. The Pane does not perform layout on its children. The position of a node that is a child of a Pane-object, is determined solely by the translate and other transformations of the child-node in question. Even though JavaFX provides much flexibility in the way transformations such as translation, scaling and rotate can be applied to nodes, much of the calculations related to drawables' coordinates in the factory-view were eventually hand coded. JavaFX's powerful transformation features, including translate, rotate and scale, are however very convenient for placing and sizing various objects on the Pane.

JavaFX scene graph combined with the reactive binding using properties, is also well suited for creating simple custom graphics that compose of other simpler shapes, text or even images. To implement a circle contained within a rectangle, for example, one could bind the dimensions and coordinates appropriately. It would then be possible to freely adjust the width and height properties of the rectangle and the circle would always be nicely centred and resized within the rectangle regardless of the dimensions of the rectangle.

The following snippet presents an example code for the circle centred within a rectangle. Variable r represents the rectangle node and variable c the circle node.

```
c.centerXProperty().bind(
    r.xProperty().add(r.widthProperty().divide(2)));
c.centerYProperty().bind(
    r.yProperty().add(r.heightProperty().divide(2)));
c.radiusProperty().bind(
    Bindings.min(r.widthProperty(),r.heightProperty()).divide(2));
```

To implement the pallet with stacks from Figure 2, it is quite straightforward to have the pallet be a container. Then the stack of boxes on the pallet can be a child of the pallet in the scene graph. Then when the pallet is moved, its children are moved also. In other words, the coordinates of the children are bound to the coordinates of the parent. This is a very simple way for drawing pallets and many other things in JavaFX, but it has its drawbacks.

The initial factory-view implementation was simply placing JavaFX nodes to a layout container Pane. However, it soon became apparent that such an approach would lead to performance problems. The findings are in line with the findings of Connors [26]. Connors recommends that keeping the scene graph as small as possible helps JavaFX performance.

4.5 JavaFX Canvas

This section introduces the JavaFX preconditions for the solution that was eventually developed to enable the drawing of thousands of TUs while keeping the scene graph size, and resulting performance, at an acceptable level. As described in the previous sections, JavaFX's scene graph and reactive property binding are powerful tools that both simplify and provide flexibility for the process of generating simple custom graphics.

On the other hand, the performance efficiency starts do degrade when the scene graph contains too many nodes. The factory-view is required to visualize possibly thousands of items simultaneously. Additionally, if all of the graphics would be developed using JavaFX nodes, one drawable item could be composed of dozens of nodes themselves. It is now clear that this would result in a scene graph containing possibly tens of thousands of nodes, which is unacceptable and not something JavaFX was designed for.

JavaFX's `Canvas` enables low-level drawing primitives that do not suffer from performance problems related to the number of nodes in the scene graph. In fact, the canvas has nothing to do with nodes, except that it is a JavaFX node itself. The drawback of using such low-level drawing utilities is that it requires more complex code. Animating a circle with a number in the centre is in many ways much simpler with basic JavaFX where the framework handles the imperative "how" and the developer can focus on the declarative "what". When operating with the low-level Canvas, one must for example write code that handles both the drawing and clearing of the framebuffer whenever something new needs to be drawn.

Bitmaps used in graphics display as part of a larger scene, are sometimes called sprites. The use of sprites dates back to early computer games and can often be implemented with low resources. Drawing sprites on the `Canvas` requires just a single method call. However, the WCS UI must support zooming in the factory view. This means that to use sprites, one should either have pre-calculated bitmaps for every possible zoom-level or tolerate, likely severe, resampling artefacts in scaled bitmaps.

4.6 Snapshotting JavaFX Nodes

The decision to use JavaFX was likely related to a desire to reduce the diversity in internal tools and to focus on the Java ecosystem. As described in the previous section, using bitmaps in JavaFX's canvas could likely solve the performance problems related to a too high number of nodes in the scene graph. There are likely many good options for automatically generating sprites for different zoom levels. These include vector graphics editors which could be scripted to produce the bitmaps for different zoom-levels. Such an approach would however require tooling and expertise outside of the Java ecosystem. It would also lead to losing the previously discussed power and flexibility present in using JavaFX to generate the required graphics. The factory-view is also a part of the WCS UI, with the rest of the application being mostly form-like. As creating the form-like parts

using idiomatic JavaFX is something that the developers will eventually do anyway, it is natural to have the factory-view developed using an approach as similar as possible.

Luckily the Node class provides a method called snapshot which enables the rendering of the node into a bitmap. Using the snapshot functionality, it is possible to find a balance between idiomatic JavaFX and low-level graphics involving frame buffers and the Canvas. The drawables can be entire developed using JavaFX nodes, but drawn in the factory-view as bitmaps on the `Canvas`.

4.7 Dynamic configurability and maintainability

As described in Section 2.2.1, JavaFX's views can optionally be developed using an XML-based notation called FXML. JavaFX's use of FXML is somewhat comparable to HTML in web browsers and much like web pages, the views can be styled using JavaFX's implementation of cascading style sheets (CSS). The scene graph can also be defined entirely in Java-code without using any FXML or annotations.

The JavaFX ecosystem includes a GUI editor called Scene Builder. Scene Builder enables a what you see is what you get (WYSIWYG) style development of FXML views. Using custom components in Scene Builder requires that the custom components are built separately and then imported to Scene Builder manually. This can be a laborious task.

The users of the WCS GUI framework, the Java-developers working with customer projects with a backend emphasis, may not be familiar with web technologies. Mostly due to this, and the challenges related to the use of Scene Builder, it was decided to favor Java-code over FXML. The use of FXML injection in JavaFX enforces separation of concerns and certain architectural component boundaries. JavaFX views written in plain Java have no such restrictions. This means that more of the responsibilities related to having clean component boundaries and keeping the codebase maintainable lie on the developer.

To keep the codebase uncoupled, new object instantiation in WCS UI is handled by Spring Framework. As the vanilla GUI should work as is in simple customer projects, it needs to be dynamically configurable. To achieve this, the backend provides class names that are used by Spring to instantiate objects and inject them to the UI.

Spring also provides many useful tools and best practices. Spring handles most of the WCS UI's object graph, not just the factory-view, and defines many aspects of the architecture. Springs philosophy of "convention over configuration" means that sane defaults for most common tasks are provided by the framework. Spring's networking components, for example, were used for the backend communication as is without much deviation from Spring's defaults.

5. IMPLEMENTATION DETAILS

This section presents in detail the technical solutions that aim to solve the performance challenges described in previous sections. Attention is also given to architectural aspects that define the maintainability of the software.

Figure 3 presents the basic drawing flow. `FactoryView` extends `Pane` and is the scene graph node that contains the various layers for bitmap drawing, regular nodes, and for event handling. The different layers are, in applicable sections, opaque in colour, and transparent to user events such as mouse clicks.

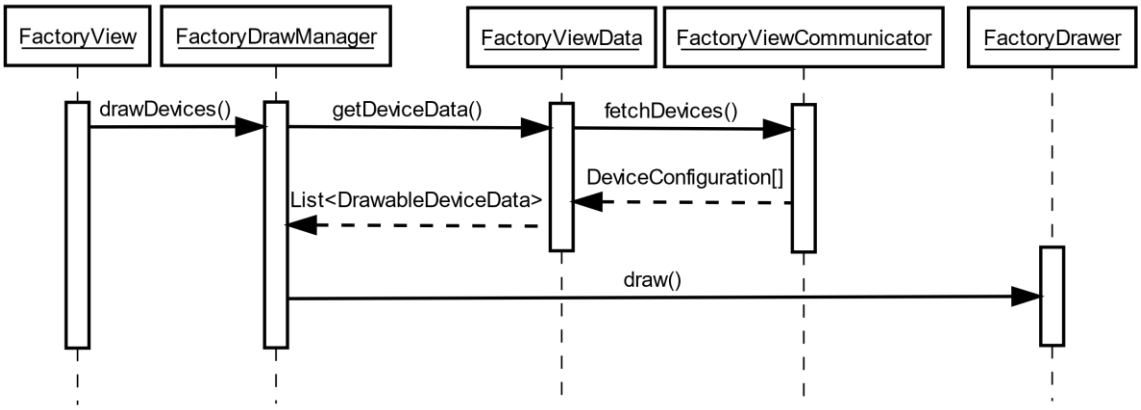

Figure 3. *Sequence diagram visualizing the program flow in drawing the devices in the factory-view.*

A WCS UI running on a client can have several factory-view instances open simultaneously. As a result, the `FactoryView` and `FactoryDrawManager` objects are specific to a single factory-view. The other classes in Figure 3 are singletons representing global shared state. Everything that is visualized in a single factory-view should be stored in `FactoryViewData`. The data is visualized by the view instances. The `FactoryDrawer` class contains no state, only static methods that generate bitmaps from instances of JavaFX Nodes that implement the Drawable interface that is described in Section 5.2.

5.1 Object lifecycle

The object lifecycle in the WCS UI is mostly handled by Spring Framework's IoC container. Objects managed by the IoC container are called beans. The instantiation and dependencies between beans are handled by the container. The WCS UI uses Spring's

Java based configuration for configuring the IoC container. The configuration files are Java classes with methods returning the bean instances. The configuration classes have annotations that are used to define the beans. The following defines a singleton bean called conveyor:

```
@Bean
public Drawable conveyor(){
    return new DrawableConveyor();
}
```

Spring beans are singletons by default. This means that only one instance of a class is used during the program lifecycle. The Java based bean definitions, like the one above, must be a part of Java class annotated with the @Configuration annotation. Also, the Spring Context must be made aware of the particular configuration class by registering it to the context.

The factory-view, and the entire WCS UI codebase can be used as is in simple projects, but the framework must be easily extendable to cater projects that require for example new drawables. Both of these viewpoints have implications on the object lifecycle.

Spring Framework enables run-time instantiation of classes that are defined as beans in the Spring Context. The backend system provides, for example, a type, size, position and orientation of a drawable. Here the type can be a bean name registered to Spring Context. A drawable can be instantiated at runtime as follows:

```
Drawable cnv = (Drawable) springContext.getBean(“conveyor”);
```

When a customer project extends the WCS UI framework, they create a new Spring Framework based Java-project. The new project will have the existing WCS UI codebase as a library dependency. This way, the project can reuse, override and extend whatever they want. If a project develops, for example, a drawable that is seen as reusable, it is trivial to pick that bean to be merged in the reference implementation.

5.2 The Drawable interface

Most things that are drawn to the factory-view implement the Drawable interface listed in Program 1. Specifically, things that need to be drawn in numbers exceeding dozens, are required to be drawn as bitmaps to enable keeping the scene graph node count as small as possible.

```
public interface Drawable{
      /** CenterX, CenterY, Width, Height, Angle */
      XYWHA xywha();
      /** zoom */
      DoubleProperty scaleProperty();
      /** unique key for bitmap-caching */
      Object getCacheKey();
}
```

Program 1. *The Drawable interface.*

The interface dictates that all instances of Drawable implement a method called xywha() that returns an object of type XYWHA. The XYWHA class defines essentially a rectangle that can be rotated around the centre point. It contains five DoubleProperty-fields for the centre point, size and angle and their getters and setters. It also contains some helper methods for cloning and setting values. XYWHA combined with a type, for example a conveyor or a crate, form the basic metadata required for drawing the real-world objects.

Even though most coordinates in JavaFX operate on the top left point, the centre point was chosen for everything to simplify the bitmap handling. The top left coordinate of a bitmap visualizing a circle, for example, is not contained within the circle. Doing all rotations around the centre points was also seen as the simplest solution.

The XYWHA describes the position, size and orientation in real world coordinates. To support zooming, the drawables must support scaling. The scaleProperty() method in the Drawable interface is intended to be bound to a global scale value. Classes that implement the Drawable interface bind the scale property via multiplication to the centre point and width and height properties.

The purpose of the Drawable interface is to provide metadata for the mechanisms that renders bitmaps from the nodes. The getCacheKey() method returns an object intended to be used as a key in a HashMap that stores already rendered bitmaps. It is not typed to provide maximum flexibility. As an example, a circle requires only a scaled radius as a cache key, whereas a rectangle would also require the angle. Returning an Object instead of just a hashKey enables to also provide a custom equals() method which provides more control over the HashMap lookup operations.

5.3 Scale manager

All of the transformations in JavaFX operate on floating point values. When storing bitmaps, a more granular approach is required. It seems that about a hundred distinct zoom values is enough to provide a visually pleasing zooming experience when the distinct

zoom values are selected appropriately and the visualized things' size ranges from 1m to 1km.

A class called `ScaleManager` was developed to provide quantized scale values. Program 2 presents the main calculation from the class. The listing demonstrates how an esthetically pleasing exponential curve for the zoom values can be calculated. The constant `NBR_OF_SCALE_VALUES` defines how many distinct zoom levels are required, typically values ranging from 50 to 200 depending on the size of the layout. The min and max scales can range from 0.01 to 0.2 depending on the layout. These are values that are configured by project developers.

```
scaleValuesArray = new double[NBR_OF_SCALE_VALUES];
final double minScaleExp = Math.log(MIN_SCALE);
final double maxScaleExp = Math.log(MAX_SCALE);
final double step = (maxScaleExp - minScaleExp) / (NBR_OF_SCALE_VALUES - 1);

for (int i = 0; i < scaleValuesArray.length; i++) {
      double exponent = minScaleExp + i * step;
      double value = Math.exp(exponent);
      scaleValuesArray[i] = value;
}
```

Program 2. *Calculating suitable scale values for different zoom levels.*

Program 2 is run when the UI starts. The `FactoryViewManager` selects a suitable scale that fits the entire factory to the screen, typically a value near the `MIN_SCALE` constant. The scale manager contains a helper method that fetches the scale index that points to a nearest scale value that is smaller than the one given as a parameter. This is useful, for example, in fitting the view contents to the view when the window is resized.

5.4 Snapshotting nodes

The `FactoryDrawManager` handles the communication between the view, event handlers and the model stored in `FactoryDrawData`. When a websocket message or a user operation triggers a draw operation, the whole view is re-drawn. The list of drawables in `FactoryDrawData` is looped through. Before any drawing is performed, the bounds including the rotation are calculated. This involves some trigonometry. The calculated bounds are then compared to the visible area in the factory-view, and if the bounds of the Drawable are outside of the visible area, the loop continues to handle the next Drawable from the list.

If the drawable needs to be drawn, the `FactoryDrawManager` compares the bounds, calculated in the previous step, to predefined size limits. If the Drawable is too big, `FactoryView`'s method `viewPortOnlyDraw()` is called. The special view-port-draw generates a suitable `SnapshotParameter` object for Node's snapshot method. This way only the visible parts are handled in the snapshot. Without such mechanisms, the GPUs texture size limit can easily be exceeded. For example, a long conveyor zoomed close could amount to tens of thousands of pixels. In cases where the drawable is only partially rendered to a bitmap, the bitmaps are always excluded from caching. When zoomed very close, there cannot be that many drawables visible. Therefore, the lack of caching in such cases causes no performance issues.

When a Drawable needs to be drawn, the `FactoryDrawer` queries the cache using the `Drawable`'s cache-key. If a bitmap is found, it draws that on the specific `Canvas` within the `FactoryView` instance. If not, a new bitmap is rendered. The Drawable is scaled and translated using the methods defined in the interface. It is then passed along with the cache and the `FactoryView` instance to the `FactoryDrawer`'s `draw()` method.

As the bitmaps are rendered only when needed, the process can be characterized as lazy. This has the drawback that the first zoom operation can feel sluggish. If this is seen as unacceptable, it should be feasible to render the required bitmaps beforehand when the UI starts.

5.5 Pixel perfect alignment of bitmaps and nodes

Large parts of the factory-view are developed using JavaFX's regular scene graph-based approach. Due to performance reasons, some utilize a Canvas based approach. As the Canvas operates with integer pixels, and the scene graph with floating point values, it requires some fine tuning to seamlessly mix bitmaps drawn on a Canvas with nodes drawn on a Pane.

The process of rendering a Drawable Node to a bitmap, begins with generating a `WritableImage` object that will hold the bitmap. The required size for the bitmap is calculated as follows. The rotated bounds are calculated and multiplied with the scale value. A small padding to hold drop shadows or other decorations are added to the dimensions. If either width or height dimension is odd, one pixel is added to make the dimensions even. Even dimensions are required for an accurate centre point.

Program 3 demonstrates the way the Drawable is rendered on the prepared `WriteableImage`. When tested with Oracle's JDK 8 on Windows 7, the calculation produces visually identical results when compared to Nodes on a Pane.

```
int minX = (int)(scaledCenterX + 0.5 – img.getWidth() / 2);
int minY = (int)(scaledCenterY + 0.5 – img.getHeight() / 2);
SnapshotParameters ssp = new SnapshotParameters();
ssp.setViewport(new Rectangle2D(minX, minY,
                                (int)img.getWidth(), (int)img.getHeight()));
ssp.setFill(Color.TRANSPARENT);
((Node) drawable).snapshot(ssp, img);
```

Program 3. *Rendering a Drawable to a WritableImage represented by img variable.*

5.6 Event handlers

Much like web browsers, JavaFX handles user events, like mouse input, via bubbling and capturing mechanisms. In the capturing phase the event is passed from the root to the event target, and in the bubbling phase it returns from the target back to the root node. The tree in this bubbling metaphor is inverted, as the "bubbles" travel from branches towards the root.

The `FactoryView` class contains various layers. Only the layers that contain too many nodes, need to utilize the bitmap snapshotting mechanism. When working with the Canvas based bitmap layers in the `FactoryView`, only the layer and the mouse coordinates are provided by JavaFX's event handlers. As we are dealing with a Canvas, determining the target Drawable that was clicked, needs to be calculated manually. This is a common requirement in computer graphics and is called the point-in-polygon problem.

To find a click target, first all possible targets can be looped through, and their rectangular bounding boxes are checked for intersection with the clicked point. If only a single match is found, that is determined to be the target. If there are several matches, all of the matches can be instantiated as nodes of their specific type. JavaFX nodes implement a method called `contains`. The `contains` method can be finally used to accurately check whether a point is contained within the shape of a node.

To handle a click which target is a Canvas, a separate class called the `EventHandlerPane` was developed. The `EventHandlerPane` initializes and sets a handler on most touch gestures and, for example, mouse and keyboard events. The class handles only input related things. Zooming, panning and tilting is forwarded to the `FactoryDrawManager` class.

If an event it is a click on a device, the handler loops through all devices from `FactoryViewData` and checks whether the mouse click coordinate is contained within one of them. To do this, the mouse click coordinate must first be transformed to real world coordinates by undoing the scaling and translating operations. If no match is found, the event is forwarded to items beneath the current Canvas. The EventHandlerPane turned out to become quite complex and as such highlights the downsides of creating UIs with low level tools.

6. ANALYSIS

This chapter evaluates the software artefact produced during this research. As described in earlier chapters, the focus of this research is twofold. First the problem domain and the software solution that was developed along this research are presented. Secondly the quality of the resulting software artefact is evaluated. The quality metrics used in the evaluation, can be narrowed down to cover performance efficiency and software maintainability. As the lead developer and architect of the software solution in question, the author of this thesis was mainly concerned with these aspects and had little control over other quality factors.

As described in Chapter 4, performance efficiency presents a key challenge in the development of the factory-view-component when using the JavaFX UI framework. Section 6.1 presents statistics that compare the performance efficiency of the chosen approach to an idiomatic JavaFX implementation. It verifies the assumption that the bitmap-based approach offers superior performance characteristics when compared to node-based approaches.

Chapter 2 argued that high performance efficiency often requires low-level or bare-metal coding approaches such as using imperative languages with manual memory management like C/C++ or even assembler. When used in conjunction with high performance graphics, such approaches can utilize low level application programming interfaces (API) such as OpenGL or DirectX that can interact with a GPU directly. In contrast, higher-level languages and approaches are often considered more maintainable. Deviating from idiomatic JavaFX, made certain compromises regarding maintainability inevitable. Section 6.2 evaluates the maintainability of the factory-view component developed during this research.

6.1 Performance characteristics

As described in Chapter 4 and Chapter 5, refresh rates below 60 Hz are generally indicative of a sub-par graphics performance in JavaFX. The refresh rate is trivially measurable using a JavaFX component called the `AnimationTimer`. A refresh rate of 60 Hz was recorded on most workloads on the final iterations of the WCS UI and the factory-view

during this research. As such it can be stated that the quality requirements regarding performance efficiency were met.

The refresh rate of the WCS-UI, and especially the factory-view, is highly dependent on different workloads. Both user actions, such as zooming, and the state of the WCS backend system have an effect on the perceived performance. To demonstrate the potential performance gains, a minimal reproducible example was developed. The complete code, excluding fairly obvious import statements, can be found from Appendix A.

Table 1. *Performance comparison results from a program listed in Appendix A*

Nbr of	**Bitmap rendering duration, ms**				**Node rendering duation, ms**			
drawables	Run 1	Run 2	Run 3	**Avg.**	Run 1	Run 2	Run 3	**Avg.**
10	1	1	1	1,0	2	2	2	2,0
100	2	2	2	2,0	16	16	16	16,0
250	1	0	1	0,7	26	30	27	27,7
500	1	0	1	0,7	27	26	25	26,0
1000	0	1	0	0,3	31	29	29	29,7
2500	1	1	1	1,0	62	71	57	63,3
5000	2	2	2	2,0	80	85	85	83,3
10000	5	4	3	4,0	96	116	96	102,7
20000	28	25	21	24,7	159	156	149	154,7

Table 1 presents numbers generated using the JavaFX application from Appendix A. The application was developed for the purpose of demonstrating the benefits of using low-level rendering in JavaFX when there is a risk of the scene-graph becoming too big. The tests were run with Oracle JDK 8 on a MacBook Pro (15-inch, 2019).

It is clear by looking at the numbers from Table 1 that rendering more than one hundred nodes is on the edge of acceptable performance. As 16 ms equates to the 60 Hz limit discussed in earlier sections, it is safe to assume that exceeding this with the hardware used for running the tests will lead to reduced frame rate and is indicative of a full utilization of a one or more CPU core causing a bottleneck. By running the code, one can also observe that rendering even hundreds of nodes causes severe memory consumption and increasing the drawable count will, eventually depending on the hardware, crash the test application.

The somewhat naive example from Appendix A and Table 1 omits some aspects. Though the numbers regarding bitmaps are good when compared to nodes, it seems that the example represents bitmap-performance in a bad light. It is likely that such a short-lived test application suffers from overhead not present in actual application, as 25 ms with 20 000 bitmaps seems slower than what is experienced with the actual application and more complex tests. Table 1 does not fail to show that the bitmap-based drawing method is significantly faster.

The test also omits performance measurements related to animating the nodes. By modifying the code from Appendix A, it is possible to verify that animation performance starts to degrade when the number of animated nodes increases. When the code is modified so that only a single iteration of nodes is added using the measure-method, and the `Canvas` related parts are removed, what is left is a view containing a random pattern of nodes.

When the code from Appendix A is modified as described above, it can be augmented with the following loop to add a separate rotate animation to each node:

```
for(Node n : p.getChildren()){
    RotateTransition rt = new RotateTransition();
    rt.setNode(n);
    rt.setByAngle(360);
    rt.setDuration(Duration.millis(10000));
    rt.setCycleCount(Animation.INDEFINITE);
    rt.play();
}
```

When the above example is run with even 250 nodes on the relatively powerful hardware described above, it can be seen by using the `htop` command, for example, that one or two CPU threads are being constantly fully utilized, the fans of the MacBook become loud, and JavaFX frames are being dropped. At around 1,000 nodes, JavaFX starts outputting internal errors. As the factory-view is required to animate possibly thousands of items, and taking into consideration the test results from this section, it is fair to assume that using idiomatic JavaFX as such would not have worked in this problem domain.

This section does not prove that the novel approach presented by this thesis is better. Including a self-contained example that draws animated bitmaps using `Canvas` would likely require several pages of code, and as such is omitted for brevity. Tests performed during the development, and experiences from using the final solution, indicate that animating several thousands of nodes using the solution developed during this thesis works very well with minimal CPU load and no frames dropped. The following section suggests

that the domain specific solution that was developed during this research, in addition to providing adequate performance, causes only a minor reduction in maintainability.

6.2 Maintainability aspects

As described in Chapter 4, expendability is an important variable when evaluating the maintainability of the WCS UI. Meyer's Open-Closed principle (OCP) [44, p. 54], is an object-oriented principle that, when applied to this context, states that it should be possible to develop new GUI functionality without modifying existing code. As the WCS UI needs to be usable as a library, adhering to the OCP is also a functional requirement.

The WCS UI uses Spring Framework to replace JavaFX's dependency injection mechanism. Spring includes functionality that enables the extension and replacement of parts of the program, for example in customer projects. Spring also provides useful tools and conventions especially related to the network-client nature of the application, but this design decision has its drawbacks.

Spring operates with Plain Old Java Objects (POJO). These are regular java classes with typical getter and setter naming conventions. The Spring Context maintains the object-graph composed of POJOs and handles POJO instantiation and lifecycle. As described in Chapter 2, this provides many benefits regarding testability and configurability. Every POJO requires a piece of configuration that brings it to the knowledge of Spring. Spring does not prevent circumventing the DI mechanism in any way, and in many cases, it is totally acceptable to instantiate non-fundamental objects directly from Java code.

When the resulting WCS UI software codebase is inspected, it seems that only some of the objects in the application are handled by Spring. Many, even large, object hierarchies are instantiated without using Spring. It is likely that such an approach will lead to project teams having to tamper with upstream code. This can cause problems with maintainability.

Earliest iterations of the factory-view did not use Spring. The architecture was also not sufficiently documented and communicated in the project team. These reasons are likely candidates for the causes resulting in the situation that Spring is not utilized accordingly. Luckily, while tedious, making regular objects Spring-managed is trivial and the situation will likely improve naturally while the software is being used and further developed.

Using Spring Framework to replace JavaFX's DI mechanism has another bigger drawback. With idiomatic JavaFX's FXML-views and injected controllers, the MVC-like style

is heavily enforced. With JavaFX mixed with Spring, and without FXML, nothing is preventing developers from interleaving presentation code with the data and business logic. This turned out to be a bigger problem than anticipated. In general, it is difficult to know how to separate business logic from the presentation. Possibly partly due to the general difficulties in separating presentation from business logic, merely communicating the importance of separating views and controllers, and providing good examples seems sometimes not to be enough. In retrospect, it might have been worthwhile to develop an abstraction that mimics the JavaFX DI as much as possible. If the views and controllers would always be instantiated by the Spring context, it could be less tempting to develop ad-hoc solutions that can lead to reduced maintainability.

The resulting factory-view component has one fly in the ointment that seemed to be impossible to fish out. The snapshotting mechanism does not perfectly fit in the JavaFX's object-oriented class hierarchy. As described in section 5.2, the `Drawable` interface is in key role when working with the snapshottable nodes. Everything that is drawn on as bitmap on the factory-view, is a subclass of `Node` that implements `Drawable`. Nodes that implement `Drawable` are however problematic. `Node` is an abstract class, not an interface. In Java, multiple inheritance is only possible using interfaces. In WCS UI, classes implementing `Drawable` are always subclasses of `Node`. When snapshotting instances of `Drawable`, an explicit typecast to `Node` is always required. This limitation may be due to JavaFX's history with JavaFX Script that supported mixins, a form of multiple inheritance. The typecasts could be avoided by overriding all nodes used in the factory-view to implement the `Drawable` interface, but few explicit typecasts here and there are likely still more maintainable.

Additionally, there exists many other DI tools in the Java ecosystem. Spring was selected because the project team had some familiarity with it, and Spring provided many useful components, such as networking and authentication, that worked well enough with the default configuration. In retrospective, other DI frameworks could have been more suitable for the task at hand. There even exists a project called Afterburner that provides JavaFX specific IoC. Using Afterburner could have possibly helped in enforcing the MVC-like division of components in JavaFX while providing all the benefits of DI.

Event handling, such as mouse clicks, zooming and panning, was quite challenging to implement and the result is not overly impressive when inspected from the aspect of maintainability. This is reflective of the challenges generally present in using low-level techniques in UI development. Quite a lot of work would be required to clean up the event

handling code. For bitmaps, all the event handling and target searching had to be implemented manually, and it needed to be interoperative with native JavaFX components. The result is a set of classes that work but are difficult to maintain. Luckily the event handling implementation has worked well enough and new requirements that would require refactoring are currently not in sight.

The maintainability of the system where the JavaFX nodes are rendered into bitmaps and viewed on canvas, is likely to cause only minor maintainability challenges for future developers. Apart from the event handling, and especially selecting the click target from the correct layer, the system just works. When new drawables are developed for the factory-view, the developers only need to adhere to the few rules posed by the `Drawable` interface described in Section 5.2. This is not much different from developing pure JavaFX nodes. As such, it can be concluded that, apart from the current complex state of the event handling system, the maintainability of the resulting factory-view is acceptable.

6.3 Limitations and future work

The resulting software framework provides a means to an end, a framework that serves its purpose in the context of the WCS UI. This means that as such, the scope of the research is limited by the domain of the WCS UI. While the resulting solution is quite flexible, no attempt was made in making it general purpose. Although this research provides most of the building blocks for a general-purpose solution, the many domain specific parts limit the usefulness of this research. These include functionality related to exceptions developed for supporting different layers in the WCS UI. These are also related to the maintainability issues with the domain specific event handling described in Section 5.6. and Section 6.2.

The node bitmap hybrid solution seems to be a novel idea in the context of JavaFX. It could be possible to develop an entirely encapsulated JavaFX layout container that would accept regular nodes as its children and would handle the snapshot rendering and altered node-object lifecycle behind the scenes while conforming to all JavaFX conventions like layout, resizing and event handling. As such, the mechanism of using JavaFX nodes as source material for automatic bitmap rendering, and the accompanying zoomable and pannable views, could be a good candidate for further research.

Instead of developing a general-purpose JavaFX layout container, it could be possible to use this research as a foundation for developing a general-purpose drag and drop style GUI builder. Such a thing could be useful in applications that need to visualize

things that do not fit on a single screen, that is, views that need to support scaling or translation, or both. Applications involving blueprints are an obvious candidate, as the software was developed for visualizing factories in a blueprint like manner. Home automation, or things like smaller heating, ventilation, and air conditioning control systems like the ones used in farms, could be a possible application area. Creating programmable graphics in a drag and drop manner could be useful in other contexts too. A utility for teaching programming or for making simple games are other possible options, especially when considering the popularity of the Java programming language.

7. CONCLUSIONS

This thesis presented a novel approach for creating maintainable code that produces high performance 2D graphics using JavaFX and no additional graphics toolkits. A related solution for performance improvement, but with a different technology stack, has been presented by Marx et al. [42] who explore 3D texture creation using web technologies. Kruk et al. present JavaFX best practices and tools including the usage of Spring Framework [35]. The paper by Kruk et al. [35] is related to more conventional form-based GUIs but can serve as a good starting point for readers interested in extending JavaFX's IoC functionalities.

The biggest contribution of this research is likely the mechanism that uses JavaFX scene graph nodes in the creation of sprites and the related functionality that allows the interoperability of sprites and nodes in a zoomable view. This thesis presents the solution at a level that should enable a developer with some familiarity with JavaFX to implement an application with comparable functionality. Most of the actual code from the resulting application is omitted, but the snippets that are presented, were carefully selected to cover the most relevant and the most challenging aspects related to the solution.

The resulting software artefact provides a means for using JavaFX for creating complex user interfaces without sacrificing much of the benefits that reactive JavaFX provides for the domain in question. The snapshot mechanisms and the views related to it also allow the developer to freely mix idiomatic scene graph-based JavaFX with the high-performance Canvas based implementations in relevant parts of the WCS UI.

It can be argued that instead of using JavaFX, it would have been a better alternative to use a game engine or even the older, and more low-level, Java GUI toolkit Swing. The solution that was implemented as part of this research does have some advantages. It does not introduce any additional tools that require learning. The viewpoint favouring small domain specific in-house frameworks is however rather subjective. It is likely that at least for some developers, it would be easier to work with a well-documented and feature rich framework. This may be especially true if the UI framework needs to be extended to meet unanticipated requirements. Three dimensional views, for example, are not supported by the implemented solution.

There are several deficiencies int the final solution when compared to more mature frameworks. The end result is not particularly well documented and the architectural

component bounds are not as strict as they could be. On the other hand, the core classes in the factory-view amount to around a thousand lines of code, so the system is small enough to be understood by experts in the field. The small size reflects the domain specific nature of the framework and may be seen as an advantage when compared to more complete external frameworks or toolkits.

JavaFX is a modern GUI framework with many similarities with contemporary web technologies. It provides features, such as transformations on nodes, that are not available in web, but the downside of the high-level approach in some applications is performance. This thesis demonstrates that it is feasible to implement complex zoomable views using JavaFX while still utilizing much of the modern reactive tools that JavaFX offers.

It was concluded that there is room for improvement regarding the maintainability of the developed solution. Replacing JavaFX's FXML based injection with the Spring Framework provided flexibility. Keeping the views and controllers separate, however requires more discipline from the developers as the separation is not in any way enforced. In conclusion it can be stated that the solution serves its purpose. At the time of writing this thesis, the performance of the WCS UI is more than adequate but there is room for improvement regarding software maintainability.

8. REFERENCES

[1] H. Abelson, G. J. Sussman, and J. Sussman, Structure and Interpretation of Computer Programs. 2nd. MIT Press. Full text available on-line. http://mitpress. mit. edu/sicp/full~…, 1996.

[2] J. Arthur and S. Azadegan, Spring framework for rapid open source J2EE web application development: A case study, *Proc. - Sixth Int. Conf. Softw. Eng., Artif. Intell. Netw. Parallel/Distributed Comput. First ACIS Int. Work. Self-Assembling Wirel. Netw., SNPD/SAWN 2005*, 2005, pp. 90–95, vol. 2005.

[3] M. Asimow, *Introduction to design*. Prentice-Hall, 1962.

[4] M. Autili, A. Di Salle, F. Gallo, A. Perucci, and M. Tivoli, Biological Immunity and Software Resilience: Two Faces of the Same Coin?, in *International Workshop on Software Engineering for Resilient Systems*, 2015, pp. 1–15.

[5] E. Bainomugisha, A. L. Carreton, T. van Cutsem, S. Mostinckx, and W. de Meuter, A Survey on Reactive Programming, *ACM Comput. Surv.*, 2013, pp. 52:1--52:34, vol. 45, no. 4.

[6] J. Bonér, F. Dave, K. Roland, and T. Martin, The Reactive Manifesto, 2014. .

[7] M. Carkci, *Dataflow and Reactive Programming Systems: A Practical Guide*, 1st ed. USA: CreateSpace Independent Publishing Platform, 2014.

[8] K. Conboy, R. Gleasure, and E. Cullina, Agile Design Science Research, in *Proceedings of the 10th International Conference on New Horizons in Design Science: Broadening the Research Agenda - Volume 9073*, 2015, pp. 168–180.

[9] G. H. Cooper, Integrating dataflow evaluation into a practical higher-order call-by-value language, *Brown Univ. Provid. RI*, 2008.

[10] O.-J. Dahl and K. Nygaard, How Object-Oriented Programming Started. [Online]. Available: http://kristennygaard.org/FORSKNINGSDOK_MAPPE/F_OO_start.html. [Accessed: 03-Dec-2018].

[11] G. E. Dieter and L. C. Schmidt, Engineering design/George E. Dieter, Linda C. Schmidt. Boston: McGraw-Hill Higher Education, 2009.

[12] E. W. Dijkstra and E. W., Letters to the editor: go to statement considered harmful, *Commun. ACM*, 1968, pp. 147–148, vol. 11, no. 3.

[13] A. Dresch, D. P. Lacerda, and J. A. V. Antunes, *Design science research: A Method for Science and Technology Advancement*. Springer, 2015.

[14] J. Edwards and Jonathan, Coherent reaction, in *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications - OOPSLA '09*, 2009, p. 925.

[15] C. Elliott and P. Hudak, Functional Reactive Animation, in *International Conference on Functional Programming*, 1997.

[16] M. Fowler, InversionOfControl, 2005. [Online]. Available: http://martinfowler.com/bliki/InversionOfControl.html. [Accessed: 15-Aug-2016].

[17] M. Fowler, GUI Architectures. [Online]. Available: https://www.martinfowler.com/eaaDev/uiArchs.html. [Accessed: 20-Jan-2019].

[18] M. Fowler, The New Methodology, 2005. [Online]. Available: https://www.martinfowler.com/articles/newMethodology.html.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns. 1995*. Reading, Massachusetts: Addison-Wesley., 1995.

[20] Google Trends, Google Trends - Spring Framework vs. Java Platform, Enterprise Edition, 2016. [Online]. Available: https://www.google.com/trends/explore?date=all&q=%2Fm%2F0dhx5b,%2Fm%2F0bs6x. [Accessed: 05-Aug-2016].

[21] T. Hawkins, Declarative programming language simplifies hardware design | EE Times, *EE Times*, 2003.

[22] S. Henty, UI Response Times, 2015. [Online]. Available: https://medium.com/@slhenty/ui-response-times-acec744f3157. [Accessed: 09-Dec-2019].

[23] A. R. Hevner, A three cycle view of design science research, *Scand. J. Inf. Syst.*, 2007, p. 4, vol. 19, no. 2.

[24] P. Hudak, Conception, evolution, and application of functional programming languages, *ACM Comput. Surv.*, 1989, pp. 359–411, vol. 21, no. 3.

[25] IEC and ISO, 25010 (2011). Systems and software engineering--Systems and software Quality Requirements and Evaluation (SQuaRE)--System and software quality models, *International Organization for Standardization, Geneva, Switzerland*. .

[26] James Connors, Node Count and JavaFX Performance, *Oracle Jim Connors' Blog*, 2009. [Online]. Available: https://blogs.oracle.com/jtc/node-count-and-javafx-performance. [Accessed: 17-Oct-2018].

[27] M. J. Jipping and K. Bruce, Imperative Language Paradigm, in *Computer science handbook*, Chapman & Hall/CRC, 2004.

[28] R. E. Johnson and B. Foote, Designing Reusable Classes, *J. Object-Oriented Program.*, 1988, pp. 22–35, vol. 1, no. 2.

[29] R. Johnson, *Expert one-on-one J2EE design and development*. John Wiley & Sons, 2004.

[30] R. Johnson, Spring Framework: The Origins of a Project and a Name, 2006. [Online]. Available: https://spring.io/blog/2006/11/09/spring-framework-the-origins-of-a-project-and-a-name. [Accessed: 15-Aug-2016].

[31] K. Kambona, E. G. Boix, and W. De Meuter, An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications, in *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, 2013, pp. 3:1--3:9.

[32] B. W. Kernighan, D. Ritchie, and others, *The C programming language*. Prentice-Hall Englewood Cliffs, 1988.

[33] S. P. Khatri, N. V Shenoy, J.-C. Giomi, and A. Khouja, Logic synthesis, in *Electronic Design Automation for IC Implementation, Circuit Design, and Process Technology: Circuit Design, and Process Technology*, CRC Press, 2016, p. 27.

[34] R. Kowalski, Algorithm= logic+ control, *Commun. ACM*, 1979, pp. 424–436, vol. 22, no. 7.

[35] G. Kruk, O. Da Silva Alves, E. Roux, and L. Molinari, Best practices for efficient development of JavaFX applications, 2018.

[36] C. Larman, *Agile and iterative development: a manager's guide*. Addison-Wesley Professional, 2004.

[37] E. A. Lee and D. G. Messerschmitt, Synchronous data flow, *Proc. IEEE*, 1987, pp. 1235–1245, vol. 75, no. 9.

[38] J. W. Lloyd, Practical advantages of declarative programming, in *Unknown*, 1994, pp. 3–17.

[39] R. C. Martin, *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall Press, 2017.

[40] R. C. Martin, The dependency inversion principle, *C++ Rep.*, 1996, pp. 61–66, vol. 8, no. 6.

[41] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*. Pearson Education, Inc., 2003.

[42] R. Marx, S. Vanhove, W. Vanmontfort, P. Quax, and W. Lamotte, DOM2AFrame: Putting the Web back in WebVR, 2017.

[43] S. McConnell, *Software estimation: demystifying the black art*. Microsoft press, 2006.

[44] B. Meyer, *Object-oriented Software Construction (2Nd Ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.

[45] B. a. Myers, Graphical User Interface Programming, in *chapter 48 of Computer Science Handbook -- Second Edition*, 2004, pp. 48-1-48–29.

[46] J. Nielsen, *Usability Engineering*. Elsevier Science, 1994.

[47] P. Norvig, Design Patterns in Dynamic Languages. 1998.

[48] L. T. O'Neil, The Latest Innovation: JavaFX, 2007. [Online]. Available:

https://web.archive.org/web/20071023073844/http://www.sun.com/featured-articles/2007-0508/javafx/index.jsp. [Accessed: 10-Oct-2019].

[49] Oracle, Java™ Platform, Standard Edition 8 API Specification, 2018. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/.

[50] J. C. Pereira and R. de F.S.M. Russo, Design Thinking Integrated in Agile Software Development: A Systematic Literature Review, *Procedia Comput. Sci.*, 2018, pp. 775–782, vol. 138.

[51] Pivotal Software Inc., Spring Framework, 2016. [Online]. Available: http://projects.spring.io/spring-framework/. [Accessed: 15-Aug-2016].

[52] R. R. Pucella, Reactive programming in Standard ML, in *Computer Languages, 1998. Proceedings. 1998 International Conference on*, 1998, pp. 48–57.

[53] A. Rauschmayer, *Speaking javascript: An in-depth guide for programmers*. O'Reilly Media, Inc., 2014.

[54] T. Reenskaug, The Model-View-Controller (MVC) - Its Past and Present, in *JAOO Conference, Aarhus, Denmark*, 2003.

[55] M. L. Scott, *Programming language pragmatics*, 4th ed. Morgan Kaufmann, 2015.

[56] N. Sculthorpe, Towards safe and efficient functional reactive programming, University of Nottingham, 2011.

[57] H. A. Simon, *The sciences of the artificial*. MIT press, 1996.

[58] R. E. Sweet, The Mesa programming environment, in *ACM SIGPLAN Notices*, 1985, vol. 20, no. 7, pp. 216–229.

[59] A. Taivalsaari, T. Mikkonen, M. Anttonen, and A. Salminen, The death of binary software: End user software moves to the web, *Proc. - 9th Int. Conf. Creat. Connect. Collab. through Comput. C5 2011*, 2011, pp. 17–23, no. 978.

[60] TIOBE Software BV, TIOBE Index | TIOBE - The Software Quality Company, 2016. [Online]. Available: http://www.tiobe.com/tiobe-index/. [Accessed: 22-Aug-2016].

[61] A. Troelsen, *Pro C# 2008 and the. NET 3.5 Platform*. Apress, 2008.

[62] R.-G. Urma, M. Fusco, and A. Mycroft, *Java 8 in action : lambdas, streams, and functional-style programming*. Manning Publications Co.

[63] I. Valkov, N. Chechina, and P. Trinder, Comparing Languages for Engineering Server Software: Erlang, Go, and Scala with Akka, 2018.

[64] G. Walls, *Spring in Action, Third Edition*. Manning Publications Co., 2011.

[65] L. B. Wilson and R. G. Clark, *Comparative programming languages*. Addison-Wesley, 2001.

[66] C. Zhang and D. Budgen, What Do We Know about the Effectiveness of Software Design Patterns?, *IEEE Trans. Softw. Eng.*, 2012, pp. 1213–1231, vol. 38, no. 5.

[67] Chris Lattner's Homepage. [Online]. Available: http://nondot.org/sabre/. [Accessed: 10-Oct-2019].

[68] Using FXML to Create a User Interface. [Online]. Available: https://docs.oracle.com/javafx/2/get_started/fxml_tutorial.htm. [Accessed: 19-Feb-2019].

[69] Hello World, JavaFX Style. [Online]. Available: https://docs.oracle.com/javase/8/javafx/get-started-tutorial/hello_world.htm. [Accessed: 10-Dec-2018].

[70] Spring Framework Documentation. [Online]. Available: https://docs.spring.io/spring/docs/current/spring-framework-reference/index.html. [Accessed: 10-Dec-2018].

APPENDIX A: JAVAFX NODE PERFORMANCE EXAMPLE

The following presents a minimal reproducible example on the benefits of using low-level bitmap drawing over JavaFX nodes in cases where the scene-graph contains too many nodes. The applications performs test runs by drawing the letter P on two different ways. It outputs three columns of data, 1. the number of items drawn, 2. time it took to draw the bitmaps in ms, and 3. time it took to draw the nodes in ms. The code uses `Platform.runLater` quite heavily. This is required to make sure that JavaFX performs the drawing, applies Cascading Style Sheet (CSS) and finishes everything before moving to the next step.

```
public class Main extends Application {
    private WritableImage img;
    private GraphicsContext gc;
    private Pane p;
    @Override
    public void start(Stage primaryStage) throws Exception{
        Parent root = perfTest();
        primaryStage.setScene(new Scene(root, WIDTH, HEIGHT));
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
    private Pane perfTest(){
        Pane root = new Pane();
        Canvas c = new Canvas(WIDTH, HEIGHT);
        this.p = new Pane();
        this.gc = c.getGraphicsContext2D();
        root.getChildren().addAll(c, p);
        Label pLabel = newNode();
        pLabel.setTextFill(Color.GREEN);
        root.getChildren().add(pLabel);
        // workaround to force layout before snapshot
        Platform.runLater(() -> {
            img = snapshot(pLabel);
            root.getChildren().remove(pLabel);
            startTest();
        });
        return root;
    }
```

Code continues on the next page …

… continued from the previous page.

```
    final static int WIDTH = 1600;
    final static int HEIGHT = 800;
    final static int BMAP_TEST = 1;
    final static int NODE_TEST = 2;
    private void startTest(){
        List<Long> bmapResults = new ArrayList<>();
        List<Long> nodeResults = new ArrayList<>();
        List<Integer> iterationsList = new ArrayList<>();
        iterationsList.add(10);
        iterationsList.add(100);
        iterationsList.add(250);
        iterationsList.add(500);
        iterationsList.add(1000);
        iterationsList.add(2500);
        iterationsList.add(5000);
        iterationsList.add(10000);
        iterationsList.add(20000);
        for(int iterations : iterationsList) {
            Platform.runLater(()->bmapResults.add(measure(iterations, BMAP_TEST)));
            Platform.runLater(()->nodeResults.add(measure(iterations, NODE_TEST)));
            Platform.runLater(()->p.getChildren().clear());
        }
        Platform.runLater(()->printResults(iterationsList, bmapResults, nodeResults));
        Platform.runLater(()->System.exit(0));
    }
    private long measure(int count, int testType){
        long start = System.currentTimeMillis();
        for(int i = 0; i < count; i++){
            double x = Math.random() * WIDTH;
            double y = Math.random() * HEIGHT;
            if(testType == NODE_TEST) drawTo(p, newNode(), x, y);
            if(testType == BMAP_TEST) drawTo(gc, img, x, y);
        }
        long finish = System.currentTimeMillis();
        return finish - start;
    }
    private Label newNode(){
        Label l = new Label("P");
        l.setFont(new Font(100));
        return l;
    }
    private void drawTo(GraphicsContext gc, Image img, double x, double y){
        gc.drawImage(img, x, y);
    }
    private void drawTo(Pane p, Node n, double x, double y){
        n.setTranslateX(x);
        n.setTranslateY(y);
        p.getChildren().add(n);
    }
    private WritableImage snapshot(Node n){
        SnapshotParameters ssp = new SnapshotParameters();
        ssp.setFill(Color.TRANSPARENT);
        return n.snapshot(ssp,null);
    }
    private void printResults(List<Integer> iterationsList,
                              List<Long> bmapResults,
                              List<Long> nodeResults) {
        for(int i = 0; i < iterationsList.size(); i++){
            System.out.print(iterationsList.get(i) + "\t");
            System.out.print(bmapResults.get(i) + "\t");
            System.out.print(nodeResults.get(i) + "\t");
            System.out.print("\n");
        }
    }
}
```