Comparison of Object-Oriented and Functional Programming for GUI Development

Eugen Kiss

M A S T E R ' S T H E S I S

Leibniz Universität Hannover

Faculty of Electrical Engineering and Computer Science

Human-Computer Interaction group

August 2014

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Acknowledgements

I like to thank the expert reviewers Jonathan Giles, Li Haoyi, Tomas Mikula and Evan Czaplicki for sanity checking parts of this thesis. A special thanks goes to Tomas Mikula whose generous feedback was invaluable.

Contents

Abstract

Traditionally, the object-oriented paradigm has been considered a good fit for designing and developing graphical user interfaces (GUIs). A large number of GUI frameworks are based on object oriented programming (OOP). Recently, functional programming (FP) has become more popular for mainstream development. This thesis is concerned with a comparison of the object-oriented approach and the functional approach applied to the development of graphical user interfaces. The languages and libraries which serve as a practical testbed for the comparison are Java/JavaFX on the object-oriented side, and Scala/ScalaFX, Scala.Rx, ReactFX and Elm on the functional side.

Chapter 1

Introduction

Even though OOP was born with the SIMULA programming language in the 1960s [15], it was Smalltalk that coupled GUI development with the OOP paradigm [25]. Smalltalk's inventor Alan Kay was inspired by SIMULA, among other influences, to create a new language for graphics-oriented applications leveraging the OOP paradigm – and the rest is history [24]. Ever since Smalltalk successfully showed the promise of OOP for GUI development a great deal of subsequent OOP languages and, particularly, object-oriented GUI toolkits came into existence and eventually into the mainstream. Nearly all mainstream programming languages today directly support the OOP paradigm and there is hardly any widely used GUI toolkit that does not use OOP. Various popular examples of toolkits like Cocoa, WinForms, Qt, wxWidgets, Tk, Swing and GTK+[1] all seem to validate the notion of OOP and GUI development being a good fit.

It is presumably not a historical accident that Alan Kay chose the OOP paradigm seeing as it turned out to be so successful[2]. Let us therefore try to understand the intuition behind why OOP and GUI programming go so well together.

- A typical GUI consists of a hierarchy of nested GUI elements or widgets, say a window consisting of a menu bar and a scroll area which itself contains labels and buttons and so on. Likewise, objects can be composed in such a way that their resulting structure reflects the structure of the GUI.
- There are usually many individual widgets which share common properties. For example, windows in a GUI application have different contents yet they all have a title bar, size and position. In OOP a class represents

[1] GTK+ even goes so far as to encode its own OOP system on top of C.

[2] There is even an interesting anecdote about Apple developers using Pascal to implement six initial deficient GUI applications for Lisa (an Apple Computer from the 1980s) and finally extending Pascal with OOP features from Smalltalk (resulting in Clascal) to finish the application in a reasonable manner [36].

the common structure of the objects that are its instances but the instances themselves can have different manifestations.

- A widget has its own attributes, state and behavior, e.g. its position or its visibility status, and so does an object. A widget provides operations that are initiated by user input, e.g. open, close, move and hide. This can be thought of as an object having methods which can be initiated by message passing.
- Often, a user of a GUI toolkit wishes to specialize a widget for his specific use case to, for instance, paint it in a different way. OOP has the concept of inheritance to offer a solution for this common wish.
- Last but not least, OOP's original motivation in SIMULA came from the need for writing simulations. From this perspective a GUI application actually can be seen as a kind of simulation which is controlled by user interactions.

As we can see, there is a relatively natural mapping between OOP concepts and GUI concepts.

Apart from the empirical and intuition-based evidence in favor of OOP for GUI programming, there is at least one additional *technical* argument by Aldrich [1]. He argues that OOP's "key technical characteristic" dynamic dispatch facilitates the creation of frameworks of which GUI toolkits are a notable form. On the other hand, there does not seem to exist a complete technical analysis of OOP features in terms of their relation to GUI development which could either mean that OOP "obviously" fits or that in principle other paradigms could fit just as well[3].

The landscape of previous research in FP and GUI development is fairly bleak. There are a couple of older and newer attempts at creating a functional GUI toolkit in Haskell, for example [7, 12, 27]. None of them seem very practical though. Interestingly, almost all of them incorporate functional reactive programming (FRP) in some way or another as an alternative for event-based programming. A recent, more promising take on FRP and web-based GUIs is the Elm language [14]. Still, the scarcity of *practical* functional GUI toolkits and previous research does not bode well for FP.

So far much speaks in favor of OOP for GUI development which is also the prevailing impression of most practitioners. Yet, there *is* reason to believe that FP principles might simplify GUI development. Layout combinators might make it easier to describe a GUI's visual structure. Higher-order functions and potentially functional reactive programming might make event-based programming easier. Pervasive immutability might make concurrency and other GUI-specific considerations like undo/redo support easier. The advantages of a transparent and natural support for concurrency should not be understated as more and more not only direct user interactions but also

[3] An anectodical sentiment from [20] is that "concurrency and delegation provide many of the same advantages as OOP".

(periodic) interactions from sensors and remote internet services court for the GUI's attention. Plus, one should not forget that while the idea of FP is as old as that of OOP only in recent years did FP gain substantive momentum in the mainstream and, thus, many of its potential areas of application are not extensively explored.

One of the main goals of this thesis is, therefore, to help fill the gap around the question of FP's suitability for GUI development. It makes sense then to compare FP to the currently dominant paradigm for GUI development in order to identify its strengths and weaknesses. Such a comparison would be meaningless for practical purposes if it would be conducted in a hypothetical sphere. This is why we will compare concrete programs written in concrete languages and environments.

The outcome of these comparisons can be manifold. It could be that OOP retains its iron grip on GUI development as it turns out to be superior to FP in most aspects. Conversely, FP might gain the upper hand as it turns out to offer better alternatives to OOP in most aspects. In fact, neither paradigm might turn out to be clearly better on the whole which could either lead to a recommendation of combining features from both paradigms – if there are no feature trade-offs, that is – or simply to a confirmation of the status quo. Putting paradigms aside, another possible outcome is that the language's, the IDE's and/or the toolkit's role in GUI development is much more crucial than the role of the paradigm.

To sum up, we want to identify the advantages and disadvantages of FP in comparison with OOP for client-side GUI development and find out if there are any differences in productivity that can be attributed to FP and OOP, respectively, as well as benefits of particular features for simplifying the development of GUIs. Moreover, we will investigate what the main concepts are that GUI developers need to grasp in order to productively create GUIs and the typical sources of error in either paradigm. Usability of programming languages is the wider research area where this work can be classified in.

This thesis is organized as follows. Chapter 2 presents related work and how this thesis differs from previous research. Chapter 3 is the main part of this thesis where implementations in both Java/JavaFX and Scala/ScalaFX of representative case studies are compared in terms of a fixed set of evaluation criteria. Chapter 4 lays the focus on FRP. First, the potential additional benefits of FRP are examined by extending the solutions for the case studies in ScalaFX with Scala.Rx [37] and ReactFX [38]. Afterwards, it is examined whether *pure* functional programming together with FRP provides further benefits or not by providing implementations of a selection of the case studies in Elm [39]. Finally, chapter 5 presents the conclusions and chapter 6 outlines open problems and further work to be done.

1.1 Characteristics of OOP and FP

Conflicting as these two paradigms appear, OOP and FP do have at least one aspect in common: neither paradigm has an authoritative, agreed-upon definition. Nonetheless, there are, without a doubt, plenty OOP and FP languages, opinions on the differences between the two paradigms and, not least, heated debates. For the sake of having a basis for discussion in the context of this thesis we are going to provide a *short* overview of each paradigm's main characteristics based on established sources and the representatives used for main evaluation, namely Java and Scala.

The lack of a unified understanding for the description of OOP gave rise to modern attempts at distilling at least the essence of objects [40, 1]. The focus of these definitions lies on the support of objects for dynamic dispatch and on an object having its behavior exported by its interface. Although cogent these definitions do not intend to provide a complete characterization of typical OOP. A more established view is given by Wikipedia where OOP is described as a "paradigm that represents concepts as 'objects' that have data fields and associated procedures known as methods" [41]. A resulting program consists then of interacting objects. Yet, objects are not the only concept to understand when dealing with typical OOP. To thus find these common concepts Armstrong analyzed a plethora of OOP literature which resulted in the following "OOP taxonomy": abstraction, class, encapsulation, inheritance, object, message passing, method and dynamic dispatch [3]. Armstrong's list is already quite complete. Still, we want to extend the list with some cultural or idiomatic observations about OOP as these aspects undeniably influence the particular style of programs: class hierarchies, many objects having mutable state and employment of design patterns.

The introduction of the term FP goes back to John Backus [4]. Backus identifies the following defining elements of FP: every syntactical construct is an expression[4], prevalent usage of higher-order functions such that a program is mostly build up of the combination of higher-order functions and a focus on immutability and avoiding side effects[5]. In short, FP emphasizes side-effect free (pure) functions that take and return immutable values or functions[6]. Another idiomatic aspect of FP is the employment of persistent data structures.

With respect to the general number of concepts as opposed to the number of GUI-specific concepts a developer needs to grasp in order to be productive we note that on the surface OOP appears to require more concepts to be

[4] This means that there is no distinction between expressions and statements as found in most non-functional languages.

[5] Backus used the terms *combining forms*, *history sensitivity* and *storage*.

[6] Over the years the notion of FP branched into different schools to encompass more ideas from later introduced functional languages. A discussion of these details, however, would go beyond the scope of this thesis.

learned than FP. This observation should be kept in mind whenever learnability is compared between the paradigms in the following parts of this thesis as OOP seems to have a higher initial cost[7].

We also observe that the distinction between FP and OOP is sometimes blurred, especially with our candidates Java and Scala. Does the existence of lambdas in Java 8 already make Java a functional language? Does the existence of classes in Scala make Scala a primarily object-oriented language? The point is that one can generally encode either paradigm in a language. The crucial question is how convenient this resulting encoding turns out to be. So even if Java has lambdas[8] this does not automatically mean that a resulting program with lambdas is already functional as there are still other characteristics of FP to follow in order to faithfully call a program functional. The same is true for the other direction. In this thesis, we encode the OOP paradigm in Java and the FP paradigm in Scala as this way the resulting encoding will be most natural.

1.2 Challenges in GUI Development

One of the main challenges in GUI development is to design a usable presentation of and interaction with the resulting graphical application [30]. In the context of this thesis we are *not* concerned with this admittedly very important aspect. We are instead concerned with GUI development from a *programming* perspective. That is, the structure of the program behind and the process of programming the resulting GUI application will be our subjects of inspection. A good overview of typical problems in GUI development from a programming perspective is provided by Karagkasidis [23]. He identifies four main problem categories: creating and assembling the GUI, handling user input, dialog control and integrating GUI and business logic. We are going to elucidate these four points as well as expand on general challenges of reactive systems of which GUIs are a particular instance.

The existence of GUI builders for nearly all popular GUI toolkits demonstrates that there are reasons for avoiding manual layout specifications in code. Of course, GUI builders have some inherent advantages[9] such as easier discovery of all the widgets, immediate design feedback for prototyping and little need to learn the layout API which in principle allows non-programmers to create the layout. Still, the question is whether the paradigm, language (environment) or toolkit can reduce the demand for a GUI builder which

[7] On the other hand, one could argue that OOP only makes the already necessary concepts explicit so we will not lend too much substance to the above observation.

[8] Even Smalltalk – probably the most OOP language – already had lambdas which were called “blocks”.

[9] Some disadvantages are finicky controls, shortcomings for complex layouts, reliance on a tool and either generated code that must not be changed manually or external layout files and missed opportunities for abstraction (cue “composability”).

would lead to simpler tooling requirements or, at least, to a convenient alternative for situations where for some reason or another a GUI builder cannot be used.

Most toolkits are event-based and thus handle user input and interactions between widgets among themselves and with other portions of the code with callbacks[10]. Especially in languages without succinct syntax for anonymous functions, e.g. Java before version 8, callbacks are coupled with boilerplate code that makes the intent of the code harder to identify. But even with succinct syntax for anonymous functions the event-based interactions or rather the control flow between the components in a reactive system is hard to figure out due to the inversion of control that callbacks bring along [28]. Among other alternatives, FRP shows promise to alleviate this problem by allowing "artificial" callbacks[11] to be replaced with explicit expressions of functional dependencies without inversion of control [34].

Dialog control is concerned with user-interactions in several consecutive steps. The challenge is to keep the state/context from one step to another between components and to communicate with the domain-specific parts of the program at appropriate points. Consider the following example of a vector drawing program to make the set of issues clearer[12]: the user right-clicks a rectangle, gets a pop-up menu and selects the properties menu item. A properties window is displayed and after adjustments are finished by clicking the OK button they are forwarded to the model which in turn notifies the rectangle to update. There are many questions regarding the implementation of such an interaction such as where the responsibilities among the components should reside. In such a scenario there will be many dependencies between the components and it is not easy to make the corresponding code easy to follow.

One of the most challenging tasks is the separation of GUI-specific and domain-specific code. This fundamental issue gave rise to the famous Model-View-Controller (MVC) pattern which – much like GUI development itself – was invented in the context of Smalltalk [42]. Even if the toolkit is not built with an MVC-like pattern in mind, in order to make complex GUIs manageable the developer will at some point be forced to separate domain-specific from GUI-specific code one way or the other. In any case, the MVC pattern or some variation/alternative thereof must be understood by the developer which requires a non-trivial upfront learning investment.

An additional issue with MVC-supporting toolkits is that we often cannot use the provided widget model directly as our domain-specific data structure might differ substantially. Therefore, we need to extend the widget model to fit our specific needs. This leads to the creation of a lot of classes if we are

[10] Callbacks, observers and listeners all represent essentially the same concept.

[11] That is callbacks that do not need to create side-effects and really only establish a functional dependency between components but do so in an inverted and implicit way.

[12] Adjusted and summarized from [23].

consequent with the MVC approach and makes the resulting code not only more complex to write but also more complex to read, maintain and refactor. If the domain-logic already exists it could be expensive to adapt it to the observer pattern by adding change notification mechanisms. If we avoid the adaption of preexisting domain-logic we need to add a layer of indirection. In the context of Java and other mainstream OOP languages XML is often used for the specification of the view. However, the introduction of another language to achieve the desired MVC separation seems to hint at a deficit in the original language and has other issues as well[13]. Ultimately, MVC and similar approaches make complex code more manageable but have their own complexity costs, too. The question is how to keep these costs as low as possible and whether a change in paradigm can help achieve lower costs.

Yet another common challenge in reactive systems is concurrency. Whenever a long running task, like the download of a picture that will be displayed in a window once it has finished loading, has to be accomplished the GUI must stay responsive. Juggling competing interactions with the GUI from direct user input but also from sensors and remote internet services is another challenge that will only gain importance in the future. To that end, the language or toolkit must provide a concurrency mechanism and depending on the implementation the provided solution might have quite some conceptual overhead.

[13] For example, whenever the layout/view has to be adjusted a context switch from the programming language to XML occurs. Due to XML's verbosity and multiple equivalent ways of encoding data, layout specifications in XML are hard to read and to create. Correctly connecting the XML layout files with the rest of the program is often somewhat involved and rigid. The main advantage of using XML is that a lot of tools work with XML, for instance GUI builders. But these tools could in principle work just as well with a tree-like (embedded) domain-specific language (DSL) in the original language.

Chapter 2

Related Work

As hinted in the introduction there is not a lot of research in terms of GUI *programming* as opposed to GUI development in a wider sense and much less in terms of OOP and FP in relation to GUI programming. Nonetheless, there does exist some more or less directly related work. In the following a short overview of this work is given with the less directly related work coming first and the more directly related work coming last. Also, work on programming language usability and GUI programming in general comes first, then GUI programming from an OOP and FP perspective, respectively, and finally work on GUI programming from *both* an OOP and FP perspective.

Most work on the usability of programming languages is of controlled experimental study nature with a focus on cognitive aspects. Such studies have high meaningfulness but are often expensive and their results are by necessity rather narrow. An alternative is an analytical approach. The Cognitive Dimensions of Notations framework (CDs) [43] is a set of standardized criteria for analyzing the usability of "information artifacts"[1]. This framework has been used in a variety of papers to analytically investigate the usability of programming language features or an API [9, 21, 26, 33]. Often, CDs is only applied insofar as it makes sense for a particular information artifact. That is, some criteria are left out and some are added. In this way CDs is taken as a basis for the evaluation criteria in chapter 3.

The paper "Developing Principles of GUI Programming Using Views" [6] from 2004 argues in favor of abstracting GUI programming from the language or toolkit and thus seeing it as an independent paradigm. The focus of the work is not on how to tackle fundamental GUI programming challenges with ideas from different paradigms but rather to provide the ability to create the GUI in a language and toolkit independent way and thus enable reuse of GUIs between different systems and a standardization of GUI notations. There is another promising approach to overcome challenges of classical event-based GUI programming beside OOP or FP, namely replacing

[1] Mostly software systems and in particular programming languages.

callback code with state machines. For example, SwingState [2] is concerned with integrating such state machines into Java/Swing and, more recently, InterState [35] combines state machines with a visual live editor to further ease GUI development especially in terms of expressing constraints.

The object-oriented perspective on GUI programming is primarily concerned with patterns – foremost variations of the MVC pattern – and their benefits in terms of managing the complexity of the resulting GUI related code. Examples include the aforementioned paper by Karagkasidis "Developing GUI Applications: Architectural Patterns Revisited" [23] or the famous book "Design Patterns" by the Gang of Four [19]. A modern and thorough overview of various design patterns for GUI programming is given by Martin Fowler in his article "GUI Architectures" [44] and by Artem Syromiatnikov and Danny Weyns in their paper "A Journey Through the Land of Model-View-* Design Patterns" [45].

Apart from the references to functional GUI toolkits given in the introduction [7, 12, 27] there are a couple of more papers and works on functional GUI toolkits. "A Multi-threaded Higher-order User Interface Toolkit" [20] from 1993 describes eXene, a functional GUI toolkit in Concurrent ML with a design that differs substantially from typical object-oriented GUI toolkits and a focus on concurrency. "Functional Languages and Graphical User Interfaces – a review and a case study" [31] from 1994 examines the difficulty of working with GUI code in a purely functional setting. "Structuring Graphical Paradigms in TkGofer" [8] from 1997 describes the implementation of several graphical programming paradigms using the GUI library TkGofer. Its principal focus is on how monads help structuring the resulting code. Another functional toolkit is sml_tk [46]. We take the opportunity to note that, naturally, many more toolkits in functional languages exist that are essentially a shim over a widespread object-oriented GUI toolkit. How functional these toolkits are ultimately depends on the toolkit itself and on how the programmer structures his code.

Functional Reactive Programming (FRP), which will be discussed in more detail in chapter 4, is a relatively recent functional approach to describe time-dependent and user interaction code. In the Haskell world the FRP GUI libraries Reactive Banana [47] and Sodium [48] are prominent. As noted in the introduction, the Elm language by Evan Czaplicki et al [39, 14] is a promising approach to pure functional web-based GUI programming and FRP. The thesis "Elm: Concurrent FRP for Functional GUIs" [13] discusses Elm's approach to functional GUI programming. Elm's merits for the case studies of this thesis are discussed in section 4.3. Two other interesting papers on FRP in relation to programming interactive systems like GUIs are "Deprecating the Observer Pattern with Scala.React" [28] by Ingo Maier and Martin Odersky and "REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications" [34] by Guido Salvaneschi et al. Both these papers show how FRP constitutes a better alternative to a lot of uses of callbacks

in event-based programming. Another similar paper "Functionally Modeled User Interfaces" [11] argues that being explicit about the data flow in GUI code makes the code clearer by comparing this approach to the traditional imperative approach in the object-oriented toolkit Java/Swing. Yet another paper that relates FRP to OOP is "Crossing State Lines: Adapting Object-Oriented Frameworks to Functional Reactive Languages" [22]. Its primary concern is the question of integrating a reactive language into an existing object-oriented GUI toolkit.

Probably the most directly related paper is "Programming Graphical User Interfaces with Scheme" [18] from 2003 that presents the widget library Biglook that combines object-oriented and classical functional styles for GUI programming. The authors' conclusion is that even in a mostly functional language like Scheme the programming style is still very imperative and that using OOP to handle graphical objects and FP (in the form of closures) to implement interface reactivity yields a more compact implementation for GUIs than most of the other graphical toolkits at the time.

More than ten years later the conclusion of the paper essentially has been incorporated into modern mainstream object-oriented toolkits like JavaFX 8 [49]. In JavaFX 8 callbacks are mostly written using lambdas instead of anonymous classes. In addition, bindings in JavaFX can be considered a special case of FRP. In this thesis we want to investigate if the conclusion of the work on Biglook still holds true or if there are yet novel areas where FP can improve upon OOP in terms of GUI programming and what happens if OOP is completely taken out of the picture[2].

[2] As Elm is purely functional without an ounce of object orientation we examine in section 4.3 the consequences of this approach.

Chapter 3

Case Studies

In this chapter we analyze the functional and object-oriented approach to GUI programming for a set of representative case studies that reflect fundamental challenges in GUI programming. To that end, seven case studies in increasing complexity are presented. For each case study an object-oriented as well as a functional solution are compared in terms of a fixed set of evaluation dimensions. First, however, we give a general overview on the matter, present the dimensions of evaluation and the employed toolkits. Finally, we draw our conclusions from the analysis of the case studies.

The case studies were selected by the following criteria. The set of case studies should be as small as possible yet reflect as many fundamental[1] challenges in GUI programming as possible. Each case study should be as simple and self-contained as possible yet not too artificial. Preferably, a case study is based on existing examples as that gives it more justification to be useful and there already will be at least one reference for comparison. Whereas in a traditional benchmark competing implementations are primarily compared in terms of their runtime and memory requirements, in this thesis the competing implementations are compared in terms of usability aspects of the underlying source code behind the resulting GUI applications.

JavaFX was selected as the object-oriented toolkit for mainly two reasons. First, it is a modern GUI toolkit so the object-oriented side in the comparison is not penalized due to outdated design decisions. Second, JavaFX is based on Java and the JVM which means that the syntax and semantics should be familiar to most readers and it is easy to use other JVM languages together with JavaFX like, for instance, Scala.

ScalaFX was selected as the functional toolkit for several reasons. First, it was the only mature functional wrapper around JavaFX at the time to our knowledge. Second, ScalaFX *is* a wrapper around JavaFX and therefore the comparison will not be dominated by unimportant toolkit differences but rather by conceptual differences. Third, Scala is a relatively popular

[1] Or typical or representative.

functional language that is syntactically not too distant from Java so that the solutions should be understandable even for readers unfamiliar with Scala and so that readers can concentrate more on the conceptual differences between the two approaches than on syntactical. It could be held against ScalaFX that it is not functional enough to accomplish a meaningful contrast for the case studies. However, as explained in section 1.1, Scala's support for OOP does not interfere with the ability to use Scala's support for FP to encode the functional paradigm. Plus, it is more practical to see what benefits FP could potentially have even if FP is used within an object-oriented GUI toolkit since the reality is that essentially all mature GUI toolkits are object-oriented. Nevertheless, an more "radical" functional approach to GUI programming will be covered by Elm in section 4.3.

With regard to the presented source code, irrelevant details are omitted. This means import and package declarations will not be shown. Definitions of obvious helper functions are omitted as well. For the longer case studies only relevant excerpts will be presented. JavaFX's `main` method is shown only once in the COUNTER case study and from thereon omitted as the `main` method is conceptually the same for all following case studies. The presented JavaFX code was sanity checked by Jonathan Giles, the current tech lead at Oracle in the JavaFX UI controls team, to have no obvious idiosyncracies.

3.1 Dimensions of Evaluation

As remarked in chapter 2 we base our dimensions on CDs. This way we use established criteria and thus make it easier to compare our results with other works based on CDs. The complete framework has 14 dimensions. The designers of CDs recommend to use a subset which fits the particular information artifact at hand. Consequently, we left out dimensions where we noticed that they did not bring substantial weight to the comparison. For each dimension we chose we give a short explanation as well as one or two brief examples of its meaning[2]. Even with this reduced set of dimensions we will at times omit a dimension in the evaluation when there was nothing particularly worthwhile to mention.

Abstraction Level *Types and availability of abstraction mechanisms*

Does the system provide any way of defining new terms within the notation so that it can be extended to describe ideas more clearly? Can details be encapsulated? Does the system insist on defining new terms? What number of new high-level concepts have to be learned to make use of a system? Are they easy to use and easy to learn?

Each new idea is a barrier to learning and acceptance but can also

[2] When we use the term system in the descriptions we mean the amalgamation of paradigm, language, toolkit and IDE.

make complex code more understandable. For example, Java Swing, the predecessor to JavaFX, employs a variation of the MVC design pattern in its general architecture and in particular for each of its widgets. Such being the case, there is a significant learning requirement to using the widgets reasonably well and often much boilerplate involved ("the system insists on defining new terms") which does not pay off for simple applications. On the other hand, for very complex applications the MVC-architecture may make the code more understandable and manageable as details can be encapsulated in the new terms "Model, View and Controller".

Another example is a function. A function has a name and, optionally, parameters as well as a body that returns a value following certain computational steps. A client can simply refer to a function by its name without knowing its implementation details. Accordingly, a function abstracts the computational process involved in the computation of a value. The learning barrier to the principle of a function is not great but it can still make a lot of code much more understandable[3] by hiding unimportant details.

Closeness of Mapping *Closeness of representations to domain*

How closely related is the notation to the result it is describing[4] resp. the problem domain? Which parts seem to be a particularly strange way of doing or describing something?

An example is the layout definition of a GUI. Languages that do not provide a way to describe the layout in a nested resp. hierarchical manner, and as such force the programmer to "linearize" the code with the introduction of meaningless temporary variables, make it hard to see how the structure of the layout definition relates to the resulting layout of the application. Not for nothing are XML-based view specifications widespread for GUI-toolkits in languages without native support for hierarchical layout expressions.

Hidden Dependencies *Important links between entities invisible*

Are dependencies between entities in the notation visible or hidden? Is every dependency indicated in both directions? Could local changes have confusing global effects?

If one entity cites another entity, which in turn cites a third, changing the value of the third entity may have unexpected repercussions. The key aspect is not the fact that A depends on B, but that the dependency is not made visible. A well-known illustration of a bad

[3] If the function has a descriptive name, that is.

[4] Dijkstra already considered this important dimension in his paper "Go To Statement Considered Harmful" from 1968: "We should do our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program and the process as trivial as possible."

case of Hidden Dependencies is the "fragile base class problem"[5]. In (complex) class hierarchies a seemingly safe modification to a base class may cause derived classes to malfunction. The IDE in general cannot help discovering such problems and only certain programming language features can help preventing them. Another example are non-local side-effects in procedures, i.e. the dependencies of a procedure with non-local side-effects are not visible in its signature.

Error-proneness *Notation invites mistakes*

To what extent does the notation influence the likelihood of the user making a mistake? Do some things seem especially complex or difficult (e.g. when combining several things)?

In many dynamic languages with implicit definitions of variables[6] a typing error in a variable name can suddenly lead to hard to find errors as the IDE cannot always point out such an error due to the language's dynamicity. Java's different calling semantics for primitive and reference types may lead to mistakes if the programmer mixes them up. Implicit `null`-initialization of variables can lead to null-pointer exceptions if the programmer forgets to correctly initialize a variable before its use.

Diffuseness *Verbosity of language*

How many symbols or how much space does the notation require to produce a certain result or express a meaning? What sorts of things take more space to describe?

Some notations can be annoyingly long-winded, or occupy too much valuable "real-estate" within a display area. In Java before version 8 in order to express what are lambdas today anonymous classes were employed. Compared to Java 8's lambdas these anonymous classes used to be a very verbose way of encoding anonymous functions especially when used in a callback-heavy setting like traditional GUI programming.

Viscosity *Resistance to change*

Are there any inherent barriers to change in the notation? How much effort is required to make a change to a program expressed in the notation?

A viscous system needs many user actions to accomplish one goal. Changing the return type of a function might lead to many code breakages in the call sites of said function. In such a case an IDE can be of great help. Creating a conceptual two-way data-binding by means of two callbacks involves more repetition than a more direct way to define such a dependency.

Commentary

This part is not so much a dimension but a place to mention everything

[5] See https://en.wikipedia.org/wiki/Fragile_base_class.

[6] That is, there is no need to precede a variable definition with, e.g., a `let` keyword.

else which is noteworthy and to give a conclusion. For instance, general observations that do not fit into the above dimensions, impressions during the development process, efficiency concerns of the resulting code and potential improvements can be addressed. In addition, the responsibilities of the other dimensions' results are assigned to the paradigm, language, toolkit and the IDE.

The other dimensions from CDs that were considered but ultimately did not bring much weight were:

- Premature Commitment *Constraints on the order of doing things*
- Role Expressiveness *The purpose of an entity is readily inferred*
- Consistency *Similar semantics are presented in a similar syntactic style*
- Hard Mental Operations *High demand on cognitive resources*
- Secondary Notation *Extra information in means other than formal syntax*

If the languages and toolkits were much more different than in our case the above dimensions would probably have been useful as well. If we had focused more on the tooling and the process of the creation of our case studies than on a comparison of the end results the following remaining dimensions would have probably been useful:

- Visibility *Ability to view components easily*
- Progressive Evaluation *Work-to-date can be checked at any time*
- Provisionality *Degree of Commitment to actions or marks*

3.2 Overview of JavaFX and ScalaFX

To quote JavaFX's documentation JavaFX is a "set of graphics and media packages that enables developers to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms" [49]. Its, for our analysis, relevant architectural features include:

- A scene graph, i.e. a hierarchical tree of nodes, with associated event handlers and application-specific state, that represent all of the visual elements of the application's user interface.
- Layout containers or panes to allow for the arrangement of the widgets within a scene graph.
- Callbacks on widgets to allow the implementation of behaviors.
- A data-binding system as a more declarative alternative for many uses of callbacks with support for binding expressions.
- Observable collection libraries which allow applications to wire user interfaces to data models, observe changes in those data models and update the corresponding UI control accordingly.

- A separate event dispatch thread.
- Trouble-free support to use other JVM languages with JavaFX which fosters experimentation.

The basic structure/scaffolding of a JavaFX resp. ScalaFX application will be shown in the first case study.

JavaFX also supports the specification of layouts in XML via FXML and optionally a GUI builder like Scene Builder. Note that neither FXML nor a GUI builder are employed in the solutions of our case studies as (complex) layout specifications are a challenge in only one case study[7] and using yet another language or tool would take away the focus from the main comparison of the conceptual differences between the two paradigms.

Another noteworthy aspect is that JavaFX is by default included in JDK/JRE 8 so its deployment is widespread and thus most probably its usage as a client side GUI toolkit will become widespread as well in the near future. In other words, our analysis will hopefully have more practical relevance than it would have been the case with an outdated or more obscure toolkit.

In addition to JavaFX's features and Scala's language features, ScalaFX has the following relevant characteristics:

- A hierarchical pattern for creating new objects and building up the scene graph.
- A more natural syntax for binding expressions.
- Easier interoperability with JavaFX than with pure Scala and JavaFX.

[7] And there the case for FXML or something similar is made.

3.3 Counter

Challenges: understanding the basic ideas of a language/toolkit and the essential scaffolding

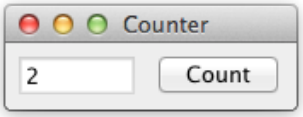


Figure 3.1: The COUNTER GUI.

The task is to build a frame containing a label or read-only text field T and a button B. Initially, the value in T is 0 and each click of B increases the value in T by one.

COUNTER serves as a gentle introduction to the basics of the language, paradigm and toolkit for one of the simplest GUI applications. Thus, by comparing COUNTER implementations one can clearly see what basic scaffolding is needed and how the very basic features work together to build a GUI application. A good solution will have very minimal scaffolding.

Listing 1 shows the solution in Java/JavaFX. In JavaFX the main class, here `Counter`, must extend the class `Application` and override the `start` method which provides the primary `stage`. In order to eventually launch the GUI application the `launch` method of the `Application` class is executed in the `main` method.

T is represented by the text field `count` and B by the button `countUp`. The task's behavior is implemented with a lambda callback for the `setOnAction` method of `countUp`. That means, whenever `countUp` is clicked the value of `count` is increased by one. Finally, `count` and `countUp` are laid out horizontally by an `HBox` with spacing and padding of 10 units and the stage is initialized.

```
public class Counter extends Application {
    public void start(Stage stage) {
        TextField count = new TextField("0");
        count.setEditable(false);
        count.setPrefWidth(50);
        Button countUp = new Button("Count");

        countUp.setOnAction(e ->
                count.setText(1+Integer.parseInt(count.getText())+""));

        HBox root = new HBox(10, count, countUp);
        root.setPadding(new Insets(10));

        stage.setScene(new Scene(root));
```

```
        stage.setTitle("Counter");
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 1: COUNTER in Java/JavaFX.

Listing 2 shows the solution in Scala/ScalaFX. The solution is very much similar to the solution in JavaFX. Instead of extending the class `Application`, `JFXApp` is extended which makes sure that the object `Counter` is automatically launched. Instead of getters and setters we can work with JavaFX properties as if they were fields. In particular, `count.text()` corresponds to `count.getText()`[8]. We see that specifying `count`'s attributes and describing the layout follows a hierarchical pattern.

```
object Counter extends JFXApp {
  val count = new TextField {
    text      = "0"
    editable  = false
    prefWidth = 50
  }
  val countUp = new Button("Count")

  countUp.onAction = (event: ActionEvent) => {
    count.text = (1 + count.text().toInt).toString
  }

  stage = new PrimaryStage {
    title = "Counter"
    scene = new Scene {
      content = new HBox(10) {
        padding = Insets(10)
        content = Seq(count, countUp)
      }
    }
  }
}
```

Listing 2: COUNTER in Scala/ScalaFX.

Evaluation

Note that this first evaluation is focused on the very basics. One reason is that the following evaluations do not have to repeat these basic observations.

[8] Without the empty pair of parantheses `count.text` corresponds to `count.textProperty()`.

Abstraction Level

The general language concepts in JavaFX's solution one mainly has to be aware of are classes and inheritance, methods and overriding, constructors, variables, getters and setters, the `main`-method as an entry point, callbacks in the form of lambdas and domain knowledge in Java's API. This sounds like a lot but if we assume the programmer is proficient in Java then the Java language related concepts required to work with basic JavaFX code are basic as well. Other than that, one must be acquainted with JavaFX and its API but this is also a given. More advanced abstraction levels are not required for Counter and the system does not insist on special abstractions so we observe that the solution in JavaFX has a low resp. good Abstraction Level.

In contrast, in ScalaFX's solution there is no need to be aware of getters and setters, and the `main`-method. However, one must be aware of singleton objects and hierarchical definitions. The concept of hierarchical definitions is very natural though. More advanced abstraction levels are not required as well so ScalaFX's solution is equally good as or slightly better than JavaFX's solution regarding the Abstraction Level.

Closeness of Mapping

The callback syntax of both solutions is very similar and indicates equally well the introduction of an anonymous function to be called when the user clicks the button. Variables are mapped to widgets which is also natural. ScalaFX's hierarchical pattern to define `count`'s attributes and the layout has a better mapping to the hierarchical nature of GUI widgets and the resulting GUI application itself but we note that in such a small example as Counter it is not critical.

Hidden Dependencies

JavaFX's solution has no obvious hidden dependencies whereas in ScalaFX's solution one must know that by setting the `stage` attribute of the enclosing singleton object the application will be implicitly shown. However, this is not a critical hidden dependency since even an absolute novice of ScalaFX will become quickly aware of this fact.

Error Proneness

In principle, not correctly overriding the `start` method of the class `Application` could lead to the GUI not being shown but the IDE and an override annotation prevent this potential problem in practice. Forgetting to set the primary stage's scene to `root` is slightly more probable in JavaFX's solution since defining and assigning the scene (`root`) happens in two different places as

opposed to in one place like in ScalaFX's solution. All in all though these are uncritical error possibilities.

Diffuseness

In the Java solution, the `main` method is needed. Its definition does not change between the case studies but it cannot be abstracted away either. In the ScalaFX solution one has to provide a type annotation for the callback parameter `e` which is not the case in the JavaFX solution. On the other hand, the types of the local variables in the ScalaFX version can be inferred and thus do not need a type annotation. Plus, there is no need for semicolons and empty parentheses on function applications can be left out. So in terms of Diffuseness ScalaFX's solution is ahead of JavaFX's.

Viscosity

As there is no[9] hierarchical way in JavaFX to set attributes or describe a layout, variable names have to be repeated when setting attributes. Although the IDE helps, e.g. in renaming attributes, it is still more noisy this way – also applies to Diffuseness. For example, the variable `root` only has to be introduced in order to set the padding attribute of the container. Also, types of variables are not inferred which leads to repetition.

Commentary

All in all the solutions are very similar. The minor differences are dominated by the toolkit(s) and to a lesser degree by the language differences. Some potential problems are in practice solved by the IDE. The different approaches of the paradigms did not come into play in such a basic example.

[9] Actually, there was support for builders in JavaFX which were only removed due to very specific technical reasons. There are other "hacky" alternatives to achieve a hierarchical pattern but they are discouraged. http://mail.openjdk.java.net/pipermail/openjfx-dev/2013-March/006725.html

3.4 Temperature Converter

Challenges: working with bidirectional dataflow, working with user-provided text input

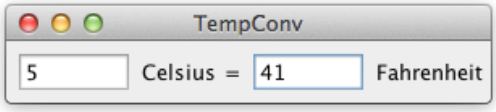


Figure 3.2: The Temperature Converter GUI.

The task is to build a frame containing two text fields T_C and T_F representing the temperature in Celsius and Fahrenheit, respectively. Initially, both T_C and T_F are empty. When the user enters a numerical value into T_C the corresponding value in T_F is automatically updated and vice versa. When the user enters a non-numerical string into T_C the value in T_F is *not* updated and vice versa.

Temperature Converter[10] increases the complexity of Counter by having a bidirectional data flow between the Celsius and Fahrenheit value and the need to check the user input for validity. A good solution will make the bidirectional dependency very clear with minimal boilerplate code for the event-based connection of the two text fields.

Listing 3 shows the solution in Java. The two text fields T_C and T_F are represented by the variables `celsius` and `fahrenheit`, respectively. The bidirectional connection between `celsius` and `fahrenheit` is accomplished by the special binding construct `bindBiderctional`. Both text properties are connected with a `StringConverter` that transforms the data flow in each direction such that the desired behavior is achieved. Finally, the layout, the scene and the stage are initialized similar to the Counter solution.

```
public class TemperatureConverter extends Application {
    public void start(Stage stage) {
        TextField celsius = new TextField();
        TextField fahrenheit = new TextField();

        celsius.textProperty().bindBidirectional(fahrenheit.textProperty(),
            new StringConverter<String>() {
                public String toString(String fahrenheit) {
                    return isNumeric(fahrenheit) ? fToC(fahrenheit) : celsius.getText();
                }
```

[10] Temperature Converter is inspired by the Celsius/Fahrenheit converter from the book "Programming in Scala" but it is such a widespread example – sometimes also in the form of a currency converter – that one could give hundred references if one liked to. The same is true for Counter.

```
                public String fromString(String celsius) {
                    return isNumeric(celsius) ? cToF(celsius) : fahrenheit.getText();
                }
            }
        );

        HBox root = new HBox(10,
                celsius, new Label("Celsius ="), fahrenheit, new Label("Fahrenheit"));
        root.setPadding(new Insets(10));

        stage.setScene(new Scene(root));
        stage.setTitle("Temperature Converter");
        stage.show();
    }
}
```

Listing 3: Temperature Converter in Java/JavaFX.

Listing 4 shows the solution in Scala which, apart from the basic differences as pointed out in the previous analysis, is essentially the same.

```
object Temperature extends JFXApp {
  val celsius = new TextField
  val fahrenheit = new TextField

  celsius.text.bindBidirectional[String](fahrenheit.text, new StringConverter[String] {
    override def fromString(c: String): String =
      if (isNumeric(c)) cToF(c) else fahrenheit.text()
    override def toString(f: String): String =
      if (isNumeric(f)) fToC(f) else celsius.text()
  })

  stage = new PrimaryStage {
    title = "Temperature Converter"
    scene = new Scene {
      content = new HBox(10) {
        padding = Insets(10)
        content = Seq(celsius, Label("Celsius ="), fahrenheit, Label("Fahrenheit"))
      }
    }
  }
}
```

Listing 4: Temperature Converter in Scala/ScalaFX.

Evaluation

We concentrate on the behavior related code as the layout and initialization code is in essence similar to the previous analysis.

Abstraction Level

The new concepts one has to know are JavaFX's properties and bindings as well as the bidirectional binding construct. These concepts are prominent features of JavaFX and similar modern GUI frameworks so it can be assumed that there is no particularly great learning barrier. From a pure language perspective anonymous classes must be known, too, though this concept is basic.

Closeness of Mapping

The special `bindBidirectional` method makes the bidirectional connection between the two text fields very clear. The transformation of the entered input in each direction is also clear due to `StringConverter`'s `fromString` and `toString` methods.

Hidden Dependencies

The special binding construct makes both directions of the dependency explicit and therefore visible.

Error Proneness

There is a minor possibility to confuse which direction of the data flow a method of `StringConverter` maps to. This can quickly be checked by trying out the application.

Diffuseness

Both solutions are very succinct.

Viscosity

Both solutions have good Viscosity as there is no unnecessary repetition.

Commentary

As before, the toolkit dominated this evaluation and the paradigms did not come into play once again. We see that a toolkit that provides special constructs for special (but not infrequent) situations improves solutions regardless of language or paradigm. Although the Abstraction Level could be improved if there was no need for a special construct this could, depending on the alternative solution, lead to penalties in the other dimensions.

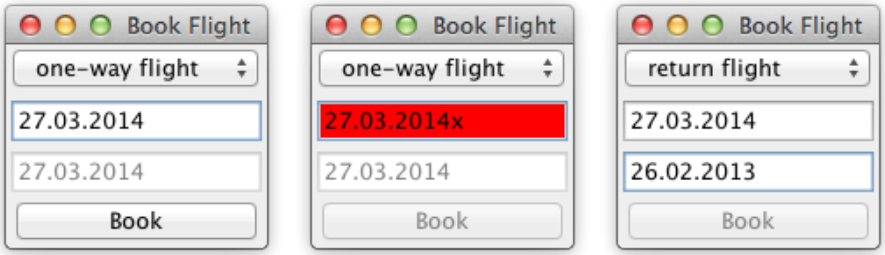
We note that if the data flow cycle would contain more than two nodes a special case solution for exactly two nodes would break down. Having a special case construct for special but frequent situations is good but an

ideal solution would also handle arbitrary cycles elegantly, e.g. by preventing feedback loops. This aspect is tested in the Cells case study (3.9).

In principle, Temperature Converter could have been solved with callbacks which would have been a worse solution though. To illustrate how worse we refer to listing 15 in the appendix, a solution in Java7/Swing that additionally does not use lambdas for callbacks. If there is no special bidirectional binding construct but only unidirectional binding constructs such a solution is also good although one has to manually take care of feedback loops. For illustration, listing 16 in the appendix shows such a solution in Clojure/Seesaw.

3.5 Flight Booker

Challenges: working with constraints



(a) (b) (c)

Figure 3.3: The Flight Booker GUI. Sub-figure (a) shows a valid one-way booking, (b) shows how an ill-formatted date disables the book button and colors the respective field red and (c) shows that the book button for a return flight is disabled if the first date is greater than the second.

The task is to build a frame containing a combo box C with the two options "one-way flight" and "return flight", two text fields T_1 and T_2 representing the start and return date, respectively, and a button B for submitting the selected flight. T_2 is enabled iff C's value is "one-way flight" (1). When C has the value "return flight" and T_2's date is strictly before T_1's then B is disabled (2). When a non-disabled text field T has an ill-formatted date then T is colored red and B is disabled (3). Initially, C has the value "one-way flight" and T_1 as well as T_2 have the same (arbitrary) date (it is implied that T_2 is disabled).

The focus of Flight Booker[11] lies on modeling constraints between widgets on the one hand and modeling constraints within a widget on the other hand. Such constraints are very common in everyday interactions with GUI applications. A good solution for Flight Booker will make the constraints clear, succinct and explicit in the source code and not hidden behind a lot of scaffolding.

Listing 5 shows the solution in Java. C is represented by `flightType`, T_1 and T_2 by `startdate` and `returnDate`, and B by `book`. The disable property of `returnDate` is bound to the value property of `flightType` to implement the first constraint. The third constraint is implemented once with a binding and once – simply for illustration – with a callback. The style property of

[11] Flight Booker is directly inspired by the Flight Booking Java example in Sodium (http://blog.reactiveprogramming.org/?p=21) with the simplification of having text fields for date input instead of specialized date picking widgets as the focus of Flight Booker is not on custom widgets.

`startDate` is bound to an expression that depends on `startDate`'s text property and will turn `startDate` red whenever its string is ill-formatted. The text field `returnDate`, on the other hand, has a callback registered on its text property such that whenever its text changes it is checked if the entered string is a valid date string. If not the field is colored red otherwise white. The second constraint is implemented with a binding that depends on the text properties of `startDate` and `returnDate` as well as the value property of `flightType`.

```
public class FlightBooker extends Application {
    public void start(Stage stage) {
        ComboBox<String> flightType = new ComboBox<>();
        flightType.getItems().addAll("one-way flight", "return flight");
        flightType.setValue("one-way flight");
        TextField startDate = new TextField(dateToString(LocalDate.now()));
        TextField returnDate = new TextField(dateToString(LocalDate.now()));
        Button book = new Button("Book");

        returnDate.disableProperty().bind(flightType.valueProperty().isEqualTo("one-way flight"));
        startDate.styleProperty().bind(Bindings.createStringBinding(() ->
            isDateString(startDate.getText()) ? "" : "-fx-background-color: lightcoral"
        , startDate.textProperty()));
        returnDate.textProperty().addListener((v, o, n) ->
            returnDate.setStyle(isDateString(n) ? "" : "-fx-background-color: lightcoral"
        ));
        book.disableProperty().bind(Bindings.createBooleanBinding(() -> {
            if (flightType.getValue().equals("one-way flight")) {
                return !isDateString(startDate.getText());
            } else {
                return !isDateString(startDate.getText()) ||
                       !isDateString(returnDate.getText()) ||
                       stringToDate(startDate.getText())
                               .compareTo(stringToDate(returnDate.getText())) > 0;
            }
        }, flightType.valueProperty(), startDate.textProperty(), returnDate.textProperty()));

        VBox root = new VBox(10, flightType, startDate, returnDate, book);
        root.setPadding(new Insets(10));

        stage.setScene(new Scene(root));
        stage.setTitle("Flight Booker");
        stage.show();
    }
}
```

Listing 5: Flight Booker in Java/JavaFX.

Listing 6 shows the solution in Scala which is similar to the solution in Java. Instead of lambdas, anonymous `Callable` instances have to be used

due to Scala 2.11's unfinished support for Java 8[12]. The operators `<==` and `===` correspond to JavaFX's `bind` and `isEqualTo` methods.

```
object FlightBooker extends JFXApp {
  val flightType = new ComboBox[String](Seq("one-way flight", "return flight"))
  flightType.value = "one-way flight"
  val startDate = new TextField{text=dateToString(LocalDate.now)}
  val returnDate = new TextField{text=dateToString(LocalDate.now)}
  val book = new Button("Book")

  returnDate.disable <== flightType.value === "one-way flight"
  startDate.style <== Bindings.createStringBinding(
    new Callable[String] { override def call(): String =
      if (isDateString(startDate.text.value)) "" else "-fx-background-color: lightcoral"
    }, startDate.text)
  returnDate.text.addListener((v: ObservableValue[_ <: String], o: String, n: String) =>
    returnDate.style = if (isDateString(n)) "" else "-fx-background-color: lightcoral")
  book.disable <==  Bindings.createBooleanBinding(
    new Callable[Boolean] { override def call(): Boolean =
      flightType.value.value match {
        case "one-way flight" => !isDateString(startDate.text.value)
        case "return flight" => !isDateString(startDate.text.value) ||
                                !isDateString(returnDate.text.value) ||
                                stringToDate(startDate.text.value)
                                    .compareTo(stringToDate(returnDate.text.value)) > 0
      }
    }, flightType.value, startDate.text, returnDate.text)

  stage = new PrimaryStage {
    title = "FlightBooker"
    scene = new Scene {
      content = new VBox(10) {
        padding = Insets(10)
        content = Seq(flightType, startDate, returnDate, book)
      }
    }
  }
}
```

Listing 6: Flight Booker in Scala/ScalaFX.

Evaluation

As before, we focus on the behavior related code, that is, the code concerned with the translation of the constraints.

[12] In principle, Scala's implicits might be used to mitigate this problem but this would drastically increase the Abstraction Level so this has not been done.

Abstraction Level

Compared to the previous two case studies there are hardly new additional concepts to learn. One aspect that might not be immediately obvious is that it is possible to create more complex bindings with the helper functions `createXBinding`, that take a function f as an argument and zero or more observables[13] such that f is reexecuted whenever one of the observables changes. A problem is that for anything but the simplest bindings one has to resort to these helper functions that are arguably more complex than a more direct way to express a binding expression like the binding expression for the first constraint.

Closeness of Mapping

From a conceptual point of view the closeness of mapping of binding expressions as opposed to callbacks for the implementation of the constraints is very good since there is no inversion of control. However, if we compare the concrete code between the binding expression approach of `startDate` and the callback approach of `returnDate` we notice that the latter approach is more terse and somewhat clearer in this case in both solutions. So the verboseness of the binding approach unfortunately obfuscates its good conceptual mapping. Ideally, the following pseudo Scala code should be all that is needed to express `startDate`'s binding:

```
startDate.style <==
  if (isDateString(startDate.text)) "" else "-fx-background-color: lightcoral"
```

Diffuseness

As remarked in Closeness of Mapping, binding expressions created with the `createXBinding` helper functions are quite verbose. An ideal solution would not even need special binding functions or constructs. Instead, native functions, operators and control structures would be reused to create binding expressions.

Hidden Dependencies

The dependencies of a binding expression created with a `createXBinding` helper function must be listed explicitly which makes them visible.

Viscosity

However, this explicit listing somewhat displaced from the function to be (re)evaluated leads to worse Viscosity. If we study the (lambda) function for the second constraint closely we see, especially in the Scala version, that

[13] JavaFX properties are observables.

the explicitly listed properties are already used inside the function. So there is a subtle repition of dependencies. Imagine the second constraint would change such that the "disabledness" of the book button would depend on the property of yet another widget. It could happen that the (lambda) function is changed according to the new requirement but explicitly listing the new property as a new dependency could be forgotten.

Commentary

Both solutions are very similar and both could be improved. The toolkit, in the form of its binding support, dominated the evaluation again. Although we saw that bindings are a better solution compared to callbacks for expressing (static) functional dependencies between widgets we also saw some current shortcomings. If the language, either natively or through libraries, provided a general and convenient way to express functional dependencies between properties of widgets or objects then, presumably, constraints two and three could have been expressed much more clearly. Chapter 4 presents such a solution.

3.6 Timer

Challenges: working with concurrency, working with competing user/signal interactions, keeping the application responsive

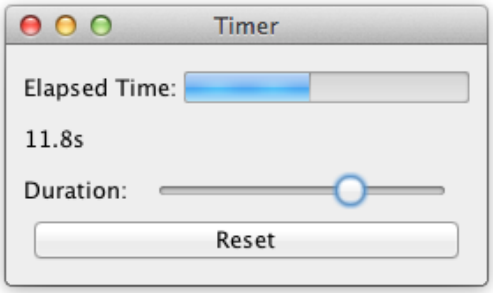


Figure 3.4: The TIMER GUI.

The task is to build a frame containing a gauge G for the elapsed time e, a label which shows the elapsed time as a numerical value, a slider S by which the duration d of the timer can be adjusted while the timer is running and a reset button R. Adjusting S must immediately reflect on d and not only when S is released. It follows that while moving S the filled amount of G will (usually) change immediately. When $e \geq d$ is true then the timer stops (and G will be full). If, thereafter, d is increased such that $d > e$ will be true then the timer restarts to tick until $e \geq d$ is true again. Clicking R will reset e to zero.

TIMER[14] deals with concurrency in the sense that a timer process that updates the elapsed time runs concurrently to the user's interactions with the GUI application. This also means that the solution to competing user and signal interactions is tested. The fact that slider adjustments must be reflected immediately moreover tests the responsiveness of the solution. A good solution will make it clear that the signal is a timer tick and, as always, has not much scaffolding.

Listing 7 shows the solution in Java/JavaFX. G is represented by `progress`, S by `slider`, R by `reset` and the numerical value of the progress is shown in label `numericProgress`. The elapsed time e is represented by the property `elapsed` and the duration d is implicitly represented by the value property of `slider` which is initialized to 20 seconds and can be between 0.1 and 40 seconds.

The progress property of `progress` – a value between zero (no progress) and one (progress completed) – is bound to the binding expression that

[14] TIMER is directly inspired by the timer example in the paper "Crossing State Lines: Adapting Object-Oriented Frameworks to Functional Reactive Languages" [22].

represents the value e/d. The text property of `numericProgress` is bound to the formatted value of `elapse` using `createStringBinding`. The callback of `reset` sets the elapsed time back to zero. The timer tick is implemented with JavaFX's animation support. That is, a time line is created that indefinitely increases `elapsed` by one every 0.1 seconds or not depending on $e < d$. At last, the layout and the stage are set up.

```
public class Timer extends Application {
    public void start(Stage stage) {
        ProgressBar progress = new ProgressBar();
        Label numericProgress = new Label();
        Slider slider = new Slider(1, 400, 200);
        Button reset = new Button("Reset");

        SimpleDoubleProperty elapsed = new SimpleDoubleProperty(0);
        progress.progressProperty().bind(elapsed.divide(slider.valueProperty()));
        numericProgress.textProperty().bind(Bindings.createStringBinding(() ->
            formatElapsed(elapsed.intValue()), elapsed));
        // For comparison, a callback based approach
        // elapsed.addListener((v, o, n) -> numericProgress.setText(formatElapsed(n.intValue())));
        reset.setOnAction(event -> elapsed.set(0));

        Timeline timeline = new Timeline(new KeyFrame(Duration.millis(100), event -> {
            if (elapsed.get() < slider.valueProperty().get()) elapsed.set(elapsed.get() + 1);
        }));
        timeline.setCycleCount(Timeline.INDEFINITE);
        timeline.play();

        VBox root = new VBox(10, new HBox(10, new Label("Elapsed Time: "), progress),
                                 numericProgress,
                                 new HBox(10, new Label("Duration: "), slider),
                                 reset);
        root.setPadding(new Insets(10));

        stage.setScene(new Scene(root));
        stage.setTitle("Timer");
        stage.show();
    }
}
```

Listing 7: TIMER in Java/JavaFX.

Listing 8 shows the solution in Scala which is similar to the solution in Java.

```
object Timer extends JFXApp {
  val progress = new ProgressBar()
  val numericProgress = new Label()
  val slider = new Slider(1, 400, 200)
  val reset = new Button("Reset")
```

```
  val elapsed = DoubleProperty(0)
  progress.progress <== elapsed / slider.value
  numericProgress.text <== Bindings.createStringBinding(
    new Callable[String] { override def call(): String =
      formatElapsed(elapsed())
    }, elapsed)
  reset.onAction = (event: ActionEvent) =>  elapsed() = 0

  val timeline = Timeline(KeyFrame(Duration(100), "", (e: ActionEvent) =>
    if (elapsed() < slider.value()) elapsed() = elapsed() + 1))
  timeline.setCycleCount(Timeline.INDEFINITE)
  timeline.play()

  stage = new PrimaryStage {
    title = "Timer"
    scene = new Scene {
      content = new VBox(10) {
        padding = Insets(10)
        content =
          Seq(new HBox(10) { content = Seq(new Label("Elapsed Time: "), progress) },
              numericProgress,
              new HBox(10) { content = Seq(new Label("Duration: "), slider) },
              reset)
      }
    }
  }
}
```

Listing 8: Timer in Scala/ScalaFX.

Evaluation

Abstraction Level

There is one new concept to understand: time line animations. This concept is straightforward to learn and it is a good abstraction for a periodic process especially compared to an explicitly thread-based alternative where the periodic aspect might have drowned in other (notational) aspects[15].

We also see that we have to use special methods (like `set`) to work with `elapsed` and cannot reuse native (like `elapsed = 0`) operations in the Java solution. That is, the system insists on using new terms instead of reusing familiar ones. In that respect, the solution in Scala is better as we can, e.g., get the value of the property `elapsed` with `elapsed()` and set it with `elapsed() = newValue`.

[15] And possibly the single-threaded nature of JavaFX would have shined through by requiring to coordinate with the special event-dispatch thread.

Closeness of Mapping

The time line approach is possibly the most natural mapping for a periodic timer tick since we as humans tend to think of time in terms of a time line.

Diffuseness

The diffuseness of both solutions is acceptable. Although we again observe that expressing bindings with `createXBinding` constructs is verbose. Using a callback base solution, like noted as a comment in the Java code, leads to terser code which really should not be the case. The terser way of getting and setting the value of a property in the Scala version is a plus.

Viscosity

If the requirements change such that the timing event is not periodic anymore but an external effect that we cannot predict the time line abstraction would fall flat. From this perspective a more general concurrency mechanism would be preferable but the toolkit or language does not provide such a mechanism to our knowledge which would also be as easy to use as the time line approach for the periodic case.

Commentary

As in the previous case studies, the toolkit dominated the evaluation. Having a special case construct for periodic events simplifies the implementation although an ideal solution would use a more general construct that is as succinct as the time line approach for the periodic case but would also withstand changing requirements.

3.7 Crud

Challenges: separating the domain and presentation logic, managing mutation, building a non-trivial layout

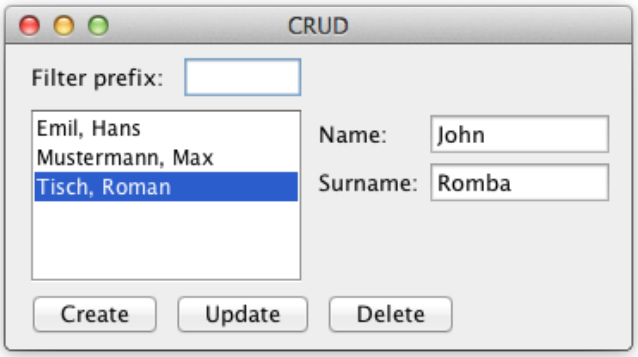


Figure 3.5: The CRUD GUI.

The task is to build a frame containing the following elements: a text field T_{prefix}, a pair of text fields T_{name} and T_{surname}, a list box L, buttons B_{C}, B_{U}, B_{D} and three labels as seen in figure 3.5. L presents a view of the data in the database that consists of a list of names. At most one entry can be selected in L at a time. By entering a string into T_{prefix} the user can filter the names whose surname start with the entered prefix – this should happen immediately without having to submit the prefix with enter. Clicking B_{C} will append the resulting name from concatenating the strings in T_{name} and T_{surname} to L. B_{U} and B_{D} are enabled iff an entry in L is selected. In contrast to B_{C}, B_{U} will not append the resulting name but instead replace the selected entry with the new name. B_{D} will remove the selected entry. The layout is to be done like suggested in figure 3.5. In particular, L should occupy all the remaining space.

CRUD[16] represents a typical graphical business application which arguably constitutes the lion's share of all GUI applications ever written. The primary challenge is the separation of domain and presentation logic in the source code that is more or less forced on the implementer due to the ability to filter the view by prefix. Traditionally, some form of MVC pattern is used to achieve the separation of domain and presentation logic. Also, the approach to managing the mutation of the list of names is tested. A good solution will have a good separation between the domain and presentation logic without much overhead (e.g. in the form of toolkit specific concepts or language/paradigm

[16] CRUD, which stands for "Create, Read, Update, Delete", is directly inspired by the article "FRP - Three principles for GUI elements with bidirectional data flow" (http://apfelmus.nfshost.com/blog/2012/03/29-frp-three-principles-bidirectional-gui.html).

concepts), a mutation management that is fast but not error-prone and a natural representation of the layout[17].

Listing 9 shows the solution in Java/JavaFX. As always, the first part consists of defining the widgets. T_{prefix} is represented by `prefix`, T_{name} and T_{surname} by `name` and `surname`, B_{C}, B_{U}, B_{D} by `create`, `update`, `delete`, and L by `entries`. The "external" database `externDb` is a list of strings. An observable copy `db` is created that is kept in sync with `externDb` by a callback on `db` that reflects its changes onto `externDb`[18]. A view `dbView` of `db` which filters `db` by a resettable predicate is created and registered as `entries`' display list. Afterwards, the actions and constraints are implemented. The properties `fullname` and `selectedIndex` are helper variables. The respective actions are registered with the widgets by callbacks and two binding expressions make sure that `delete` and `update` are enabled if and only if an item is selected. Finally, the layout is specified and the stage initialized.

```
public class CRUD extends Application {
    public void start(Stage stage) {
        TextField prefix = new TextField();
        prefix.setPrefWidth(60);
        TextField name = new TextField();
        name.setPrefWidth(100);
        TextField surname = new TextField();
        surname.setPrefWidth(100);
        Button create = new Button("Create");
        Button update = new Button("Update");
        Button delete = new Button("Delete");
        update.setDisable(true);
        delete.setDisable(true);
        ListView<String> entries = new ListView<>();
        entries.getSelectionModel().setSelectionMode(SelectionMode.SINGLE);

        List<String> externDb = new ArrayList<>();
        externDb.add("Emil, Hans");
        externDb.add("Musterman, Max");
        externDb.add("Tisch, Roman");
        ObservableList<String> db = FXCollections.observableArrayList(externDb);
        db.addListener((ListChangeListener<String>) c -> {
            while (c.next()) {
                if (c.wasReplaced()) externDb.set(c.getFrom(), c.getAddedSubList().get(0));
                else {
                    if (c.wasAdded()) externDb.add(c.getAddedSubList().get(0));
                    if (c.wasRemoved()) externDb.remove(c.getFrom());
```

[17] Note that a layout builder is not employed as it would on the one hand increase the conceptual overhead and on the other hand steer the focus away from discovering the differences between the paradigms FP and OOP although in principle a layout builder *could* be used.

[18] We do not assume a directly observable "external" database as in a real case of application this assumption most certainly would be false.

```
                }
            }
        });
        FilteredList<String> dbView = db.filtered(item -> true);
        entries.setItems(dbView);

        StringExpression fullname =
            surname.textProperty().concat(", ").concat(name.textProperty());
        IntegerExpression selectedIndex = entries.getSelectionModel().selectedIndexProperty();
        prefix.textProperty().addListener((v, o, n) ->
            dbView.setPredicate(item -> item.startsWith(n)));
        create.setOnAction(e -> db.add(fullname.get()));
        delete.setOnAction(e -> db.remove(dbView.getSourceIndex(selectedIndex.get())));
        update.setOnAction(e ->
            db.set(dbView.getSourceIndex(selectedIndex.get()), fullname.get()));
        delete.disableProperty().bind(selectedIndex.isEqualTo(-1));
        update.disableProperty().bind(selectedIndex.isEqualTo(-1));

        BorderPane root = new BorderPane();
        root.setPrefSize(400, 400);
        root.setPadding(new Insets(10));
        HBox top = new HBox(10, new Label("Filter prefix: "), prefix);
        top.setPadding(new Insets(0, 0, 10, 0));
        top.setAlignment(Pos.BASELINE_LEFT);
        root.setTop(top);
        root.setCenter(entries);
        GridPane right = new GridPane();
        right.setHgap(10);
        right.setVgap(10);
        right.setPadding(new Insets(0, 0, 0, 10));
        right.addRow(0, new Label("Name: "), name);
        right.addRow(1, new Label("Surname: "), surname);
        root.setRight(right);
        HBox bottom = new HBox(10, create, update, delete);
        bottom.setPadding(new Insets(10, 0, 0, 0));
        root.setBottom(bottom);

        stage.setScene(new Scene(root));
        stage.setTitle("CRUD");
        stage.show();
    }
}
```

Listing 9: CRUD in Java/JavaFX.

Listing 10 shows the solution in Scala/ScalaFX. Despite some superficial differences the code for the `CRUD` object is very similar to the code of the Java solution's `CRUD` class except for the layout specification. In the Scala version the layout is described in a hierarchical manner.

```
object CRUD extends JFXApp {
  val prefix = new TextField() { prefWidth = 60 }
```

```
val name = new TextField() { prefWidth = 100 }
val surname = new TextField() { prefWidth = 100 }
val create = new Button("Create")
val update = new Button("Update") { disable = true }
val delete = new Button("Delete") { disable = true }
val entries = new ListView[String]()
entries.getSelectionModel.selectionMode = SelectionMode.SINGLE

val externDb = ArrayBuffer[String]("Emil, Hans", "Mustermann, Max", "Tisch, Roman")
val db = FXCollections.observableArrayList(externDb.asJava)
db.addListener(new ListChangeListener[String] {
  override def onChanged(c: Change[_ <: String]): Unit = {
    while (c.next) {
      if (c.wasReplaced()) externDb.update(c.getFrom, c.getAddedSubList.get(0))
      else {
        if (c.wasAdded()) externDb.append(c.getAddedSubList.get(0))
        if (c.wasRemoved()) externDb.remove(c.getFrom)
      }
    }
  }
})
val dbView = db.filtered(new Predicate[String] {
  /* Irrelevant details omitted */
  override def test(t: String): Boolean = true
})
entries.setItems(dbView)

val fullname = surname.textProperty.concat(", ").concat(name.textProperty)
val selectedIndex = entries.getSelectionModel.selectedIndexProperty()
prefix.text.addListener((v: ObservableValue[_ <: String], o: String, n: String) =>
  dbView.setPredicate(new Predicate[String] {
    /* Irrelevant details omitted */
    override def test(t: String): Boolean = t.startsWith(n)
  }))
create.onAction = (event: ActionEvent) => db.add(fullname.get)
delete.onAction = (event: ActionEvent) => db.remove(dbView.getSourceIndex(selectedIndex.get))
update.onAction = (event: ActionEvent) =>
    db.set(dbView.getSourceIndex(selectedIndex.get), fullname.get)
delete.disable <== selectedIndex === -1
update.disable <== selectedIndex === -1

stage = new PrimaryStage {
  title = "CRUD"
  scene = new Scene {
    content = new BorderPane() {
      padding = Insets(10)
      prefWidth = 400
      prefHeight = 400
      top = new HBox(10) {
        padding = Insets(0,0,10,0)
        alignment = Pos.BASELINE_LEFT
        content = Seq(new Label("Filter prefix: "), prefix)
      }
      center = entries
```

```
        right = new GridPane() {
          padding = Insets(0,0,0,10)
          hgap = 10
          vgap = 10
          addRow(0, new Label("Name: "), name)
          addRow(1, new Label("Surname: "), surname)
        }
        bottom = new HBox(10) {
          padding = Insets(10,0,0,0)
          content = Seq(create, update, delete)
        }
      }
    }
  }
}
```

Listing 10: CRUD in Scala/ScalaFX.

Evaluation

Abstraction Level

A new toolkit-specific concept are observable and filtered collections whose principle is quite straightforward to understand and useful. That is, `entries` receives the observable and filtered view `dbView` of `db` and from now on whenever `dbView` is changed, for example by one of the callbacks, `entries` is automatically notified to refresh, `db` is changed accordingly and in turn automatically updates `externDb`. We can understand the process as a small change propagation.

The separation of GUI and domain specific code is made much clearer with observable collections, almost transparent. There is no need to introduce heavyweight concepts like the MVC pattern in this case which shows observable's collections ability to simplify code.

Closeness of Mapping

When the layout specification becomes more complex we see that having a hierarchical way to express the layout of the resulting application in code significantly improves notational clarity. Scala's solution is therefore much better in this regard[19].

Hidden Dependencies

The dependency between `db` and `dbView` is very clear. The dependency between `externDb` and `db`, on the other hand, could be clearer since it

[19] We could have used FXML for the Java solution but then the Abstraction Level would have been worsened and the Diffuseness would not have improved due to XML's verbosity.

is established by a callback and as such the dependency is not explicitly expressed but implicitly established in the body of the callback. Although probably not possible for all cases, a way to automatically lift a conventional collection to an observable one would definitely improve the solutions.

Error Proneness

The advantage of using an observable list is that one does not have to remember to fire update notifications after changes to the database which is a real plus in terms of Error Proneness. However, the explicit callback to keep `externDb` and `db` in sync is somewhat intricate and depends on very specific toolkit knowledge. Again, an automatic elevation of a conventional collection to an observable one would reduce the potential for errors.

Diffuseness

The widget definitions in the Scala version are terser and clearer due to ScalaFX's hierarchical expressions. The same is true for the layout specification. On the other hand, the current Scala version has deficits when it comes to using lambda expressions for Single Abstract Method (SAM) interfaces.

Viscosity

Changing the layout in the Scala version is easier since the correct place to introduce the according change is identifiable faster due to the better Closeness of Mapping of hierarchical layout expressions.

Commentary

From a toolkit-specific perspective observable collections are a real win. A way to automatically make an observable collection out of a conventional one would be the icing on the cake. From a language perspective hierarchical expressions are a win as well. From a paradigm perspective these hierarchical expressions are easier to implement in functional languages as these inherently emphasize composability although not strictly functional languages can support such expressions as well[20].

[20] Even Java/JavaFX before version 8 had a way to express hierarchical layouts. These were primarily removed for a very specific technical reason: http://mail.openjdk.java.net/pipermail/openjfx-dev/2013-March/006725.html

3.8 Circle Drawer

Challenges: implementing undo/redo functionality, custom drawing, implementing dialog control (i.e. keeping the context between successive GUI operations)

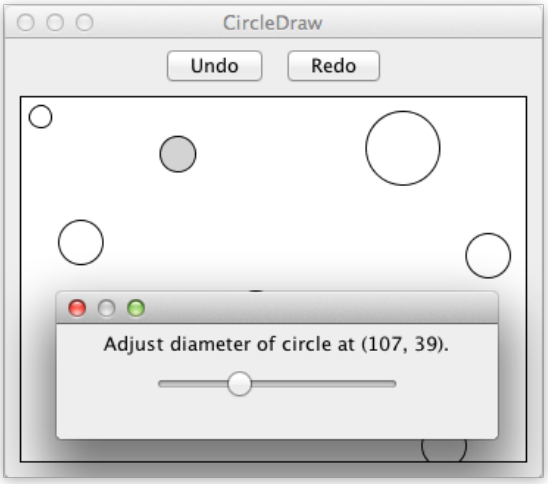


Figure 3.6: The Circle Drawer gui.

The task is to build a frame containing an undo and redo button as well as a canvas area underneath. Left-clicking inside an empty area in the canvas will create an unfilled circle with a fixed diameter whose center is the left-clicked point. The circle nearest to the mouse pointer such that the distance from its center to the pointer is less than its radius, if it exists, is filled with the color gray. The gray circle is the selected circle C. Right-clicking C will make a pop-up menu appear with one entry "Adjust diameter...". Clicking on this entry will open a dialog with a slider inside that adjusts the diameter of C. Changes are applied immediately. Closing this frame will mark the last diameter as significant for the undo/redo history. Clicking undo will undo the last significant change (i.e. circle creation or diameter adjustment). Clicking redo will reapply the last undo-ed change unless new changes were made by the user in the meantime.

Circle Drawer's goal is, among other things, to test how good the common challenge of implementing an undo/redo functionality for a GUI application can be solved. In an ideal solution the undo/redo functionality comes for free resp. just comes out as a natural consequence of the language, toolkit or paradigm. Moreover, Circle Drawer tests how dialog control[21], i.e. keeping the relevant context between several successive GUI interaction

[21] Dialog control is explained in section 1.2.

steps, is achieved in the source code. Last but not least, the ease of custom drawing is tested.

Listing 11 shows the solution in Java/JavaFX. The `CircleDrawer` class is a thin shim that initializes the undo and redo buttons as well as the heart of this case study, the class `CircleDrawerCanvas` which represents C, and contains the usual scaffolding.

`CircleDrawerCanvas` has the fields `circles` that keeps the user-created circles, `hovered` that points to the selected circle or null and `undoManager` that is responsible for the undo/redo behavior. Next, the fields are initialized and the context menu is created. The mouse interactions as described in the task are implemented, too. The `draw` and `getNearestCircleAt` methods should be obvious. The `showDialog` method is concerned with showing the diameter dialog after the entry in the selected circle's context menu has been clicked, correctly connecting the appearing slider's value with the selected circle and updating the undo history. The method `addCircle` is called whenever the user creates a new circle. Apart from adding a new circle to `circles` the corresponding undo-able command is added to the history. The methods `undo` and `redo` simply delegate to the `undoManager`. The following two inner classes `CreateCircleEdit` and `ChangeDiameterEdit` represent the respective undo-able user actions and follow the Command Pattern. Finally, the class `Circle`, `UndoManager`[22] and `UndoableEdit` are shown.

```
public class CircleDrawer extends Application {
    public void start(Stage stage) {
        Button undo = new Button("Undo");
        Button redo = new Button("Redo");
        CircleDrawerCanvas canvas = new CircleDrawerCanvas();
        undo.setOnAction(e -> canvas.undo());
        redo.setOnAction(e -> canvas.redo());
        /* Layout, scene and stage are constructed. */
    }
}

class CircleDrawerCanvas extends Canvas {
    private List<Circle> circles;
    private Circle hovered;
    private UndoManager undoManager;

    public CircleDrawerCanvas() {
        super(400, 400);
        circles = new ArrayList<>();
        hovered = null;
        undoManager = new UndoManager();

        Button diameterEntry = new Button("Diameter...");
```

[22] The implementation of the class `UndoManager` is omitted as it does not increase the value of the comparison and could have principally been provided by the toolkit.

```
        Popup popup = new Popup();
        popup.getContent().add(diameterEntry);

        diameterEntry.setOnAction(e -> {
            popup.hide();
            showDialog(hovered);
        });
        setOnMousePressed((MouseEvent e) -> {
            if (e.isPrimaryButtonDown() && hovered == null) {
                addCircle(new Circle((int) e.getX(), (int) e.getY()));
                getOnMouseMoved().handle(e);
            }
            if (popup.isShowing()) popup.hide();
            if (e.isPopupTrigger() && hovered != null)
                popup.show(this, e.getScreenX(), e.getScreenY());
        });
        setOnMouseMoved((MouseEvent e) -> {
            hovered = getNearestCircleAt((int) e.getX(), (int) e.getY());
            draw();
        });
        draw();
    }

    void draw() {
        GraphicsContext g = getGraphicsContext2D();
        /* Some details omitted. */
        for (Circle c : circles) {
            int offset = c.getDiameter() / 2;
            if (c.equals(hovered)) {
                g.setFill(Color.LIGHTGRAY);
                g.fillOval(c.getX()-offset, c.getY()-offset, c.getDiameter(), c.getDiameter());
            }
            g.strokeOval(c.getX()-offset, c.getY()-offset, c.getDiameter(), c.getDiameter());
        }
    }

    public Circle getNearestCircleAt(int x, int y) {
        Circle circle = null;
        double minDist = Double.MAX_VALUE;
        for (Circle c : circles) {
            double d = Math.sqrt(Math.pow(x - c.getX(), 2) + Math.pow(y - c.getY(), 2));
            if (d <= c.getDiameter()/2 && d < minDist) {
                circle = c;
                minDist = d;
            }
        }
        return circle;
    }

    void showDialog(Circle selected) {
        Stage dialog = new Stage();
        Slider slider = new Slider(10, 50, selected.getDiameter());

        slider.valueProperty().addListener((v, o, n) -> {
```

```
            selected.setDiameter(n.intValue());
            draw();
        });
        int oldDiameter = selected.getDiameter();
        dialog.setOnCloseRequest(e -> undoManager.addEdit(
                new ChangeDiameterEdit(selected, oldDiameter, selected.getDiameter())));

        /* dialog is constructed and shown. */
    }

    void addCircle(Circle circle) {
        undoManager.addEdit(new CreateCircleEdit(circle));
        circles.add(circle);
    }

    public void undo() { undoManager.undo(); draw(); }
    public void redo() { undoManager.redo(); draw(); }

    private class CreateCircleEdit extends UndoableEdit {
        private Circle circle;
        CreateCircleEdit(Circle circle) { this.circle = circle; }
        public void undo() { circles.remove(circle); }
        public void redo() { circles.add(circle); }
    }

    private class ChangeDiameterEdit extends UndoableEdit {
        private Circle circle;
        private int oldDiameter, newDiameter;
        ChangeDiameterEdit(Circle circle, int oldDiameter, int newDiameter) {
            this.circle = circle;
            this.oldDiameter = oldDiameter;
            this.newDiameter = newDiameter;
        }
        public void undo() { circle.setDiameter(oldDiameter); }
        public void redo() { circle.setDiameter(newDiameter); }
    }
}

public class Circle {
    private int x, y, d;
    public Circle(int x, int y) { this.x = x; this.y = y; d = 30; }
    /* Getters and setters are omitted. */
}

public class UndoManager {
    /* The implementation is omitted. */
}

abstract public class UndoableEdit {
    abstract public void undo();
    abstract public void redo();
}
```

Listing 11: Circle Drawer in Java/JavaFX.

Listing 12 shows the solution in Scala/ScalaFX. As always, many parts are almost identical to the solution in Java which is why these parts are omitted. In the following we will only go into the differences to the Java solution. The list `circles` is persistent and there is no `undoManager`. The method `showDialog` is implemented differently due to `circles` immutability. The semantic difference will be exposed in the evaluation. The implementation of the undo/redo history is done without the Command Pattern. The `history` is simply a list of previous circle lists. The variable `historyCursor` is a pointer into `history` which is needed for the correct implementation of the step-wise undo and redo user actions. If the user undoes an action the pointer is decreased and the current circle list replaced with an according previous list. The explanation for redo is dual. Whenever the user does a new undo-able action all the history states that come after the current pointer's position are thrown away and replaced with the circle list that resulted from the user's actions. Finally, the case class `Circle` is shown[23].

```
object CircleDrawer extends JFXApp {
  /* Identical to Java solution. */
}

class CircleDrawerCanvas extends Canvas(400, 400) {
  var circles = Seq[Circle]()
  var hovered: Circle = null

  val diameter = new Button("Diameter...")
  val popup = new Popup()
  popup.content.add(diameter)

  diameter.onAction = (e: ActionEvent) => /* Identical to Java solution. */
  onMousePressed = (e: MouseEvent) => /* Identical to Java solution. */
  onMouseMoved = (e: MouseEvent) => /* Identical to Java solution. */
  draw()

  def draw() = /* Identical to Java solution. */

  def getNearestCircleAt(x: Double, y: Double) = /* Identical to Java solution. */

  def showDialog(selected: Circle) {
    val dialog = new Stage()
    val slider = new Slider(10, 50, selected.d)

    val index = circles.indexOf(selected)
    slider.value.addListener((v: ObservableValue[_ <: Number], o: Number, n: Number) => {
      circles = circles.updated(index, selected.copy(d=n.doubleValue))
      draw()
    })
    dialog.onCloseRequest = (e: WindowEvent) => addSnapshot()
```

[23] For us a case class is here simply a struct, i.e. a class with value and not referenece semantics.

```
    /* dialog is constructed and shown. */
  }

  def addCircle(circle: Circle) {
    circles = circles :+ circle
    addSnapshot()
  }

  var history = Seq[Seq[Circle]](Seq())
  var historyCursor = 0

  def addSnapshot() = {
    history = history.take(historyCursor+1) :+ circles
    historyCursor += 1
  }

  def undo() = if (historyCursor > 0) {
    historyCursor -= 1
    circles = history(historyCursor)
    draw()
  }

  def redo() = if (historyCursor < history.size-1) {
    historyCursor += 1
    circles = history(historyCursor)
    draw()
  }
}

case class Circle(x: Double, y: Double, d: Double = 30)
```

Listing 12: Circle Drawer in Scala/ScalaFX.

Evaluation

Abstraction Level

The Command Pattern, as embodied in `CircleDrawerCanvas`es inner classes `CreateCircleEdit` and `ChangeDiameterEdit`, is an advanced OOP concept that has to be understood for the Java solution in order to comprehend its undo/redo implementation. On top of that, the system insists on defining new command classes for operations that should be undo-able. These command classes are not always trivial to implement either since they need to contain the relevant context to undo and redo an operation[24].

The FP solution in Scala makes do without the need for a design pattern. On the other hand, the concept of persistent data structures must be understood in order to make sense of the solution's undo/redo implementation.

[24] Even "worse", in order to not break encapsulation, the Command Pattern is often accompanied by another pattern: the Memento Pattern [19], thus further increasing the conceptual overhead.

Persistent data structures, however, and the awareness of the difference between identity and value types is a more basic FP concept than the Command Pattern in OOP.

Closeness of Mapping

There are many situations where the difference between mutable and persistent[25] data structures makes a difference in convenience and understandability – sometimes in favor of mutability and sometimes in favor of persistence.

If we consider the `showDialog` code of the FP solution we see that we must regain the index in the circle list or rather the identity of the selected circle in order to be able to observably change its parameter. Note the use of the term *observably* in the previous sentence. We do not actually change anything but the reference of the variable `circles` to a newly created circle list whose entry at the previously regained index is replaced with an updated copy of the selected circle (with a new value for its diameter). The intuitive understanding of directly changing the diameter of the selected circle is not reflected in the FP solution but it *is* reflected in the OOP solution.

Let us pretend that we wanted to connect the slider's value with the selected circle's diameter using a binding expression instead of a callback[26]. It is perfectly clear how to adapt the the OOP solution due to its closeness to our intuitiveness: just bind the selected circle's diameter property to the slider's value property. Yet, how to adapt the FP solution is not obvious.

On the plus side, the persistent nature of the circle list has a good mental mapping to the undo/redo history of the application since previous versions of the circle list are retained and they essentially reflect the undo states. The list of undo/redo operations of the OOP solution is somewhat less intuitive and more abstract.

Hidden Dependencies

It is not immediately clear when solely looking at the initialization of the variable `hover` that its value directly depends on the mouse position and the circle list. The problem is that this dependency is established spatially apart from `hover`'s initialization[27] in the callback definition of `setOnMouseMove`. It would be better to express this dependency in the initialization of `hovered` without a callback, something akin to the pseudo code

[25] Or immutable which is a special case of persistent.

[26] As a matter of fact, this was not done since we would have needed to make a circle's diameter a JavaFX property which would have involved too much overhead compared to the current solution but conceptually would have been cleaner.

[27] And putting the `setOnMouseMove` definition right after `hover`'s initialization only helps slightly since the spatial separation still exists, albeit less severe, and it would conceptually be better if there was no need for spatial separation whatsoever.

```
hovered = { getNearestCircle(mouse.x, mouse.y) }
WhenChanged(hovered) { draw() }
```

where the brackets denote that their inner expression should be interpreted specially. This is a deficit of both solutions.

Error Proneness

There is a minor possibility for error in the Scala solution compared to the Java solution. Namely, in the callback of the slider the variable `circles` has to be reassigned the new copy of the circles list which might be forgotten.

The Command Pattern is much more error prone than the alternative FP solution. First, more code is needed to implement an undo-able operation and, second, the code is more intricate thereby increasing the attack surface for mistakes.

Diffuseness

The undo/redo implementation in the Scala solution is much more succinct than the Java solution.

A basic language difference is that in Scala there is no need to introduce getter or setter methods which, for example, leads to a terser definition of the methods `draw` and `getNearestCircleAt`[28].

Viscosity

Let us suppose that the ability to drag and drop the circles is a new requirement to be implemented by us. Aside from having to implement the concrete drag and drop user interaction in both solutions (which would be almost identical) we need to make this operation undo-able. Therefore, in the Java version we need to introduce a new command class, make sure it is correct and wire it up accordingly. In the Scala version we would simply need to put one call to the method `addSnapshot` to the end of the drop operation to make drag and drop undo-able.

Let us now suppose that the created circles should be saveable so that a user can close the application and open the file he was working on sometime later. By necessity a snapshot of the relevant application state must be created and saved to disk. With the Scala approach all the hard work is already done whereas with the Command Pattern approach a lot of additional functionality has to be implemented.

[28] This is a minor point.

Commentary

This time the paradigm differences clearly dominated. In general[29], the Scala solution has the upper hand due to its undo/redo implementation which has better Viscosity, Diffuseness, Error Proneness and Abstraction Level. There is a (minor) trade-off in terms of Closeness of Mapping, though.

It is conceivable to come up with a hybrid solution that does not have the trade-off in Closeness of Mapping although one has to make sure that the circle list or rather the relevant application state is deeply copyable. In the hybrid solution `circles` is a mutable list and so the issue with the slider value does not come up. Whenever a snapshot is taken, though, an immutable deep copy of `circles` is placed into the history list. In principle, this solution could be made as (memory) efficient as the Scala solution. It would probably constitute a best of both worlds solution[30].

[29] If, however, memory consumption is a real problem even with structural sharing of the data structures and limitation of the levels of undo then the Command Pattern approach probably must be taken regardless.

[30] An implementation of this hybrid approach can be found in the implementation project of this thesis. See chapter A.

3.9 Cells

Challenges: implementing change propagation, customizing a widget, implementing a more authentic/involved GUI application

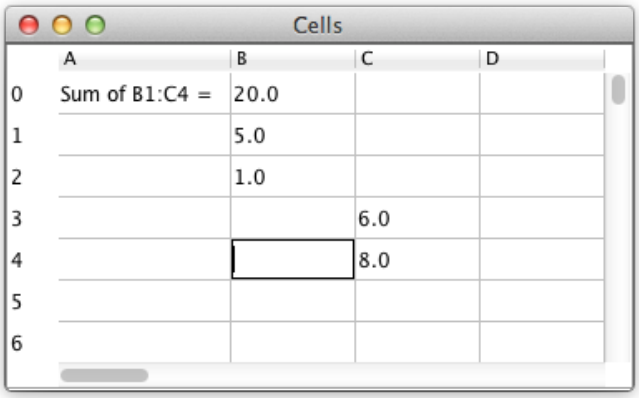


Figure 3.7: The CELLS GUI.

The task is to create a simple but usable spreadsheet application. The spreadsheet should be scrollable. The rows should be numbered from 0 to 99 and the columns from A to Z. Double-clicking a cell C lets the user change C's formula. After having finished editing, the formula is parsed and evaluated and its updated value is shown in C. In addition, all cells which depend on C must be reevaluated. This process repeats until there are no more changes in the values of any cell (change propagation). Note that one should not just recompute the value of every cell but only of those cells that depend on another cell's changed value. If there is an existing spreadsheet widget it should not be used. Instead, another similar widget should be customized to become a reusable spreadsheet widget.

CELLS[31] is a more authentic and involved task that tests if a particular approach also scales to a somewhat bigger application. The two primary GUI-related challenges are intelligent propagation of changes and widget customization. Admittedly, there is a substantial part that is not necessarily very GUI-related but that is just the nature of a more authentic challenge. A good solution's change propagation will not involve much effort and the customization of a widget should not prove to be too difficult. The domain-specific code is clearly separated from the GUI-specific code. The resulting spreadsheet widget is reusable.

Listing 13 shows the solution in Java/JavaFX. The role of the class `Cells`

[31] CELLS is directly inspired by the SCells spreadsheet example from the book "Programming in Scala". The details with respect to the parsed language can be found there.

is simply to instantiate the class `SpreadSheet` and show the application. The class `SpreadSheet`, which represents C, is the reusable spread sheet widget. It has a model, an instance of the class `Model`, and a tabular view `table` whose cells are backed by the model. The remaining code for the construction of the widget is deliberately left out as it is *very* toolkit specifc and there would be no insights to gain from it. The class `Model` (and `Cell`) represents the domain model in the MVC sense. It simply retains a two-dimensional array of cells where each cell has metaphorically two sides: one side contains the entered formula which can be empty and is shown whenever the user starts editing a cell, the other side contains the value of the cell which is shown otherwise.

`Model`'s inner class `Cell` is the heart of the application and contains the implementation of a spread sheet cell. It is both observable, as the value of other cells may depend on its value, and an observer, as its value may depend on the value of other cells. Extending the class `Observable` provides `Cell` with the necessary scaffolding to be observed (e.g. a list of observers). Implementing the interface `Observer` forces `Cell` to create an `update` method which is called by an observed cell whenever it changes. The string `userData` keeps the string for this cell exactly as entered by the user. The field `formula` contains then the abstract representation of `userData` which is initially empty. The field `value`, as the name implies, contains the numerical value of the cell. Being a non-static inner class, a cell has an implicit reference to its enclosing model (`Model.this`) which is needed to identify this cell's dependent cells. If the formula represents a textual value this textual value is the cell's string representation otherwise it is its numerical value.

The interesting part are the methods `setFormula`, `setValue` and `update`. As the formula changes, the formula's dependents change as well in general. This is why the old observers are deleted and the new observers that result from finding all referenced cells by the new formula are added. In addition, the cell's value must be updated with the help of the private helper method `setValue`. The method `setValue` stops whenever no change in the cell's value can be observed which is important to stop the change propagation at the appropriate point in time. Otherwise, the new value is adopted and the cell's observers are notified of this change. The `update` method stems from the interface `Observer` and it is called on all the observers of this current cell inside the `notifyObservers` call. Note this potentially spiral or wave-like interaction with `setValue` and `update`. Herein lies the change propagation implementation.

A sketch of the interpreter pattern for the formula representation is given for illustrative purposes which is however not relevant for the evaluation. The parsing details are omitted because of the same reason.

```
public class Cells extends Application {
```

```
    public void start(Stage stage) {
        stage.setScene(new Scene(new SpreadSheet(100, 26)));
        /* The rest is as usual. */
    }
}

public class SpreadSheet extends HBox {
    public SpreadSheet(int height, int width) {
        super();
        Model model = new Model(height, width);

        TableView<ObservableList<Model.Cell>> table = new TableView<>();
        table.setEditable(true);
        table.setItems(model.getCellsAsObservableList());

        /* The details of constructing the spread sheet are omitted. */
    }
}

class Model {
    private Cell[][] cells;

    Model(int height, int width) {
        cells = new Cell[height][width];
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                cells[i][j] = new Cell(this);
            }
        }
    }

    public Cell[][] getCells() { return cells; }

    public ObservableList<ObservableList<Cell>> getCellsAsObservableList() {
        /* Implementation is omitted. */
    }

    class Cell extends Observable implements Observer {
        private String userData = "";
        private Formula formula = Formula.Empty;
        private double value = 0;

        public String toString() {
            if (formula instanceof Textual) {
                Textual textual = (Textual) formula;
                return textual.value;
            }
            return String.valueOf(value);
        }

        public void setUserData(String s) { userData = s; }
        public String getUserData() { return userData; }

        public Formula getFormula() { return formula; }
```

```
        public void setFormula(Formula formula) {
            for (Cell cell : this.formula.getReferences(Model.this)) {
                cell.deleteObserver(this);
            }
            this.formula = formula;
            for (Cell cell : this.formula.getReferences(Model.this)) {
                cell.addObserver(this);
            }
            setValue(this.formula.eval(modelReference));
        }

        public double getValue() { return value; }
        private void setValue(double value) {
            if (!(this.value == value || Double.isNaN(this.value) && Double.isNaN(value))) {
                this.value = value;
                setChanged();
                notifyObservers();
            }
        }

        public void update(Observable o, Object arg) {
            setValue(formula.eval(Model.this));
        }
    }
}

abstract class Formula {
    public static final Formula Empty = new Textual("");
    public double eval(Model env) { return 0.0; }
    public List<Model.Cell> getReferences(Model env) { return Collections.emptyList(); }
}
class Textual extends Formula { /* Profit */ }
class Number extends Formula { /* 10 */ }
class Coord extends Formula { /* A1 */ }
class Range extends Formula { /* A1:B7 */ }
class Application extends Formula { /* =sum(A1, B2:C8) */ }

class Parser { /* The implementation is omitted. */ }
```

Listing 13: Cells in Java/JavaFX.

Listing 14 shows the solution in Scala/ScalaFX. Apart from superficial differences the relevant parts of the solution in Scala are identical to the solution in Java as the *classical* FP paradigm, as described in section 1.1 of the introduction, did not provide a satisfying alternative. The reason is mainly that we cannot avoid the Observer Pattern for an efficient implementation of the change propagation.

```
object Cells extends JFXApp {
  stage = new PrimaryStage {
    /* ... */
```

```
    scene = new Scene(new Spreadsheet(100, 26))
  }
}

class Spreadsheet(height: Int, width: Int) extends HBox() {
  val cellModel: Model = new Model(height, width)
  val table = new TableView[ObservableList[Model.Cell]]()
  table.editable = true
  table.setItems(cellModel.cellsAsObservableList)

  /* The details of constructing the spread sheet are omitted. */
}

class Model(val height: Int, val width: Int) extends Evaluator with Arithmetic {
  val cells = Array.ofDim[Cell](height, width)
  for (i <- 0 until height; j <- 0 until width) cells(i)(j) = new Cell(i,j)

  def cellsAsObservableList: ObservableList[ObservableList[Cell]] = {
    /* Implementation is omitted. */
  }

  case class Cell(row: Int, column: Int) extends Observable with Observer {
    var userData = ""
    private var f: Formula = Empty
    private var v: Double = 0

    override def toString = formula match {
      case Textual(s) => s
      case _ => value.toString
    }

    def formula: Formula = f
    def formula_=(f: Formula) = {
      for (c <- references(formula)) c.deleteObserver(this)
      this.f = f
      for (c <- references(formula)) c.addObserver(this)
      value = evaluate(f)
    }

    def value: Double = v
    def value_=(w: Double) {
      if (!(v == w || v.isNaN && w.isNaN)) {
        v = w
        setChanged()
        notifyObservers()
      }
    }

    override def update(o: Observable, arg: scala.Any) = {
      value = evaluate(formula)
    }
  }
}
```

```
/* Parsing and formula evaluation implementation omitted although different. */
```

Listing 14: CELLS in Scala/ScalaFX.

Evaluation

Abstraction Level

One must be aware of the MV* concept seeing as the solutions explicitly use a model in the MV* sense. Because of this and due to the change propagation implementation, the Observer Pattern in particular must be understood. These are advanced concepts.

The toolkit-specific domain knowledge in order to create a spread sheet widget from a table view is extremely steep. The code is not shown here due to its complexity.

Closeness of Mapping

Intuitively, we understand that each formula has dependent cells. The Observer Pattern has a good mapping to our understanding seeing as dependents are modeled as a list of observers.

Hidden Dependencies

The implementation of the change propagation mechanism could be more explicit since it requires (implicit) knowledge that `notifyObservers` calls the `update` method of an observer and it is needed to call `setChanged` beforehand.

Error Proneness

The explicit management of the change propagation, e.g. by needing to remove old observers or by the mutual calls of the methods `setValue` and `update`, invites mistakes as the implementation is somewhat intricate. Ideally, change propagation would be taken care of by the system and not the programmer.

Diffuseness

The table view customizations to make it a spread sheet are very verbose.

Viscosity

Let us say that we want to prevent feedback loops now. It is not immediately obvious how to implement this requirement. In an ideal system, feedback loops would be taken care of automatically for us.

Commentary

As there were practically no differences between the two solutions we can only discuss potential improvements.

The toolkit could make it easier to use its widgets for customization. Granted, we have probably encountered a situation that the widget was not designed for. Still, in an ideal toolkit customization should be easier.

Having to manually take care of the change propagation is intricate. It would be better if the programmer was freed from this responsibility[32].

[32] In fact, it is possible to use JavaFX Bindings to shift most of the change propagation concerns into the framework. See appendix B for a sketch. The reason this alternative has not been used in the evaluation is that we would have needed to introduce another library (EasyBind).

3.10 Verdict

Let us first briefly recap the conclusions of each case study.

- Counter showed that tiny programs are dominated by the toolkit.
- Temperature Converter showed that bidirectional data flow between two widgets can conveniently be covered by a special toolkit construct.
- Flight Booker showed that toolkit provided binding expressions are a more natural and more explicit alternative to callbacks for formulating functional dependencies between widgets.
- Timer showed yet again that having a toolkit provided special case construct for periodic events simplifies the implementation of this case study.
- Crud showed that toolkit provided observable collections greatly simplify the implementation of a typical data entry and data update GUI application. Furthermore, it was shown that a hierarchical way to describe the layout of a GUI application is beneficial.
- Circle Drawer showed that the functional paradigm with its focus on immutable values and persistent data structures greatly simplifies the implementation of an undo/redo feature.
- Cells in particular showed that there do exist shortcomings that are neither solved by the toolkit nor by the classical OOP or FP paradigm. Conveniently expressing change propagation was a challenge of Cells that could potentially be solved in a better way.

We see that the toolkit dominated the first four case studies. In the fifth case study, the part concerned with specifying the layout was dominated by language/paradigm differences but the rest was dominated by the toolkit. In the sixth case study the paradigm differences clearly dominated. And in the seventh case study no single part in particular dominated.

In the introduction we noted that our analysis could result in many possible outcomes in terms of the role of the toolkit, paradigm and language. Now that the analysis has been completed we can conclude that the toolkit in general played the most crucial role in our case studies. A good toolkit can cover potential deficits in the paradigm and/or language with special constructs for special but frequent use cases. Paradigm and language differences played a more minor role. Still, there were situations where they lead to different results.

In chapter 2 we wondered if the conclusions of the Biglook paper were still true today or if there were novel areas except for lambda expressions where FP could improve GUI application code when used in conjunction with a primarily object-oriented toolkit. After our analysis we can conclude that the classical FP paradigm provides further benefits for GUI applications that need to

keep past configurations resp. states around for e.g. undo/redo functionality or serialization. FP's focus on pervasive immutability and persistent data structures makes the implementation of the aforementioned features much easier. FP languages by their nature facilitate hierarchical layout expressions, too.

With the above conclusions in mind we want to give a recommendation of features for GUI toolkits and for languages to improve general aspects of GUI application programming. The following are thus features of JavaFX which we found especially helpful for the case studies:

- Widgets being objects from a class hierarchy and inheritance for customization[33]
- Binding expressions for describing functional dependencies between widgets as an alternative to callbacks
- Observable collections for freeing the programmer from manual change management
- Special constructs for special but frequent use cases[34]

And these are language and paradigm features which turned out to be advantageous for the case studies:

- Succinct anonymous functions for clearer expression of callbacks
- Hierarchical layout expressions for a clearer and more natural way to specify a GUI application's layout
- Persistent data structures and support for immutability in order to have easier snapshot functionality

Since the toolkit we used for the case studies is in its core still object-oriented, that is its widget hierarchy is class based and so on, we did not discover advantages or disadvantages of an out-and-out functional toolkit approach. This is not a problem for three reasons. First, our focus lay mainly on the implementation of GUI *applications* and not on GUI toolkits. Second, almost all practical GUI toolkits are in their core object-oriented so using an object-oriented toolkit made the analysis more practical, too. Third, we will still explore an approach to GUI application programming without the help of an object-oriented toolkit in section 4.3 of chapter 4.

Almost all case studies exhibited leverage points for further improvements. For example, the data binding expressions in Flight Booker were somewhat diffuse, the toolkit provided special case constructs would probably not sustain more complex interactions and the change propagation implementation of Cells was not ideal. There is reasonable hope that the next chapter will reveal novel solutions for these shortcomings.

[33] This structure of toolkits is tried-and-tested and our analysis did not uncover any problems with it.

[34] Like bidirectional bindings between two widgets and time line animations.

Chapter 4

Functional Reactive Programming

Functional Reactive Programming (FRP) is widely regarded as a promising approach to improve GUI programming in the FP community. At the same time, FRP is still a relatively young paradigm and hence not enough experience applying it to GUI programming has been gathered. For this reason, we will examine in this chapter the consequences of FRP for our case studies as an alternative to the classical callback approach of object-oriented GUI toolkits. We first classify the meaning of the term "FRP" and afterwards investigate the question why, despite FRP's potential benefits, its usage in practice is not more widespread. Finally, in sections 4.1, 4.2 and 4.3 we analyze the reactive approach based on our case studies and further examples.

Before diving into FRP we should first understand what the term FRP actually encompasses for, as we will see, "functional reactive" has neither a definite nor a unique meaning just like the attributes object-oriented and functional. Wikipedia describes reactive programming as "a programming paradigm oriented around data flows and the propagation of change" [50]. This description already gets to the core of what reactivity in the context of programming means. Now, the additional attribute "functional" in FRP as opposed to just reactive programming can be interpreted in two different ways. It can simply refer to a "programming paradigm for reactive programming using the building blocks of functional programming" [51] or it could refer to a similar but more restrictive model of FRP as used in languages like Fran [17]. We take the former interpretation in this thesis. We also note that FRP libraries differ from "non-functional" reactive libraries in that the former almost always provide functional combinators to manipulate reactive constructs and strive to be more declarative in general.

Having just outlined the difference between "non-functional" reactive programming and FRP there is still a need to understand the different schools that exist among FRP. In general, one can differentiate between FRP systems

that focus on time-varying values on the one hand and ones that focus on event streams, observable collections and asynchrony on the other hand. Scala.Rx[1] and Elm are examples for the former kind of FRP systems and Reactive Extensions (.NET Rx) [52] as well as RxJava [53] and ReactFX [38] are examples for the latter kind. Naturally, both being FRP systems there is overlap – for example in the form of functional combinators. A relatively recent and very thorough overview of the various models of FRP and their usage in languages is given in the paper "A Survey on Reactive Programming" [5][2]. The paper describes the first kind of FRP systems as "Siblings of FRP" since these are more or less directly inspired by the work in Fran [17]. The second kind is called "Cousins of FRP" as they, foremost, do not have primitive abstractions for the representation of time-varying values. We sum up that in this thesis we understand FRP as a paradigm around data flows, change propagation and time-varying values or event-streams using functional building blocks.

Now that we have a better understanding of the terminology, we briefly recap the relevant parts of section 1.2 with respect to the question of what FRP actually claims to bring to the table in terms of improving program clarity and productivity. Classical callback-based event-handling for many GUI-related use cases has downsides due to its inversion of control and required manual management of state changes and data dependencies which is error-prone [10]. This problem is known as "Callback Hell" in the programming community [16]. Data-binding systems like found in JavaFX mitigate these problems but are often limited in their abstraction capabilities. FRP is a solution to these problems[3]. FRP can even simplify the MVC pattern by essentially removing the observer pattern from MVC [54].

One could argue that JavaFX is already quite a reactive toolkit especially compared to its predecessor Swing. JavaFX has observable collections somewhat similar to those in .NET Rx and it has an imperative data-binding system that can be seen as a special case of FRP focused on automatic change propagation. Specifically, JavaFX's interface `ObservableValue` in fact represents a time-varying value. Thus, the direction of modern object-oriented GUI toolkits like JavaFX but also web-based GUI frameworks like Knockout [55] and many more seems to validate the notion that reactive programming, at the very least in the form of data-binding and observable collections, is indeed beneficial. Yet, the term "reactive" is not used by these toolkits or frameworks – why is that? Essentially, the question of how reactive a language or library is comes down to how convenient it is to define reactive

[1] Please note that despite the "Rx" suffix Scala.Rx has not much to do with .NET Rx or RxJava but much more with Scala.React [28].

[2] Another good and even more recent overview is [60].

[3] That is not to say that callbacks do not have their place in GUI programming. If an action should be executed primarily for its side effects then callbacks are a perfectly fine abstraction [34].

constructs and that enough abstraction capabilities are provided by the language and/or library. The authors of "Deprecating the Observer Pattern with Scala.React" [28] express the core issue elegantly:

> Imperative data-binding systems such as JavaFX or Adobe Flex provide a way to bind expressions to variables. This is similar to programming with signals in Scala.React. Without a first-class notion of time-varying values and events, however, those systems often lack important abstraction mechanisms and need to revert to inversion of control for complex event logic.

With the previous observations in mind let us try to understand why the reactive approach is not more widespread in practice and why modern frameworks are midway to the reactive approach yet still do not embrace it fully. First, there is a lack of syntactical convenience for expressing reactive constructs in most mainstream languages. Second, it could be that efficiency concerns and concerns in terms of conceptual overhead of (functional) reactive programming discourage toolkit implementers. Third, toolkit implementers and users could simply find (functional) reactive programming to not be worth the trouble and find the current approach good enough or even better[4]. Finally, even though substantive research on FRP started around 1997, only recently did accessible and practical FRP libraries come into existence[5]. To further illuminate this important point we quote Scala.Rx's author who gets to the heart of the matter:

> FRP is a well studied field with a lot of research already done. Scala.Rx builds upon this research, and incorporates ideas from the following projects: FlapJax, Frappe, Fran. All of these projects are filled with good ideas. However, they are generally very much research projects: in exchange for the benefits of FRP, they require you to write your entire program in an obscure variant of an obscure language, with little hope inter-operating with existing, non-FRP code.

The reasons might be much more trivial even. For example, perhaps toolkit implementers and users are simply not thoroughly aware of (functional) reactive programming. In any case, we hope to highlight at least some of the practical consequences of FRP for GUI development in this chapter.

[4] There are indeed some criticisms of FRP in particular (not reactivity in general) which we will go into in the conclusion of this chapter.

[5] The landscape for "non-functional" reactive libraries does not seem to be at this point yet. Nevertheless, we will mention some of the research in this direction later in this thesis.

4.1 Scala.Rx

Scala.Rx [37] is a change propagation library for Scala that provides reactive variables which auto-update when the values they depend on change. One of the main benefits that sets Scala.Rx apart from other FRP works is that Scala.Rx is easy to use and inter-operable. The code snippets in this section were sanity checked by Li Haoyi, the author of Scala.Rx.

The following shows an exemplary program from Scala.Rx's site.

```
val a = Var(1)
val b = Var(2)
val c = Rx{ a() + b() }
println(c()) // 3
a() = 4
println(c()) // 6
```

We observe that updating the value in `a` automatically updates the value in `c` as `c` is dependent on `a`. The underlying implementation is efficient. For instance, assigning the value 4 once more to `a` would not lead to a recomputation of `c`. Apart from the `Var` construct that creates an input node in the data flow graph and the `Rx` construct, a signal expression, that creates an inner node in the data flow graph, Scala.Rx provides an `Obs` construct that performs some side effect when the observed value changes and combinators to allow the transformation of reactive variables.

The challenge is to integrate Scala.Rx with ScalaFX. That is, we need to somehow connect the world of JavaFX resp. ScalaFX properties with the world of Scala.Rx reactive variables in order to find out if Scala.Rx can provide advantages for our case studies. Fortunately, Scala has the concept of *Implicit Classes* which basically allows to extend the functionality of a class retroactively resp. without changing the class itself. The following sketch shows how the integration is done.

```
object RxIntegration {
  val observers: Buffer[Obs] = Buffer()
  implicit class PropertyExtensions[T,J](p: Property[T,J]) {
    def rx(): Rx[T] = {
      val v = Var(p.value)
      p.addListener { v.update(p.value) }
      return v
    }
    def |=(x: => T) {
      val rx = Rx{x}
      observers += Obs(rx) { p() = rx() }
    }
  }
}
```

Each JavaFX property gains an `rx` method that wraps the property in a `Var`. Whenever the property changes the value in the respective `Var` is updated.

Furthermore, the `|=` operator is introduced. On the left hand side `|=` expects a JavaFX property and on the right hand side an expression `x` that is passed in with call-by-name semantics. This expression `x` is wrapped in an `Rx` and whenever this reactive value changes the respective property is updated. The list of observers is simply there to prevent the garbage collector from collecting the observers.

Unfortunately, the integration is not perfect. For example, expressing a binding between two ScalaFX properties using the integration looks something as follows.

```
widget_1.property_x |= widget_2.property_x.rx()()
```

Note the `rx()()`. Ideally, the binding expression would look similar to the introductory example of Scala.Rx.

```
widget_1.property_x |= widget_2.property_x()
```

In principle, achieving this is indeed possible but would require a more involved integration[6]. We acknowledge the current shortcoming of this proof-of-concept integration but want to pretend to have a mostly ideal integration in the following comparisons simply to focus more on conceptual differences.

Let us at first consider Flight Booker (3.5). The main remaining problem from the comparison in chapter 3 was that the expression of functional dependencies using `createXBinding` constructs was verbose and contained some repetition. For better illustration we show the relevant part of listing 6 once more. We even pretend that Scala already had perfect SAM conversion. That is, instead of anonymous classes we use lambda syntax[7].

```
returnDate.disable <== flightType.value === "one-way flight"
startDate.style <== Bindings.createStringBinding(
    () => if (isDateString(startDate.text.value)) "" else "red"
  , startDate.text)
returnDate.style <== Bindings.createStringBinding(
    () => if (isDateString(returnDate.text.value)) "" else "red"
  , returnDate.text)
book.disable <==  Bindings.createBooleanBinding(
    () => flightType.value.value match {
      case "one-way flight" => !isDateString(startDate.text.value)
      case "return flight" =>
        !isDateString(startDate.text.value) || !isDateString(returnDate.text.value) ||
        stringToDate(startDate.text.value).compareTo(stringToDate(returnDate.text.value)) > 0
    }
  }, flightType.value, startDate.text, returnDate.text)
```

[6] For an example of a syntactically ideal integration (albeit in a slightly different context) see the Scala.Rx TodoMVC example: https://github.com/lihaoyi/workbench-example-app/tree/todomvc

[7] We will also treat all the other Scala excerpts as if Scala already had perfect interoperability with Java 8 simply to not distract the comparison with irrelevant details.

Using our Scala.Rx integration the following is what we achieve.

```
returnDate.disable |= flightType.value() == "one-way flight"
startDate.style  |= { if (isDateString(startDate.text()))  "" else "red" }
returnDate.style |= { if (isDateString(returnDate.text())) "" else "red" }
book.disable |= {
  flightType.value() match {
    case "one-way flight" => !isDateString(startDate.text())
    case "return flight" =>
      !isDateString(startDate.text()) || !isDateString(returnDate.text()) ||
      stringToDate(startDate.text()).compareTo(stringToDate(returnDate.text())) > 0
  }
```

On the first glance we notice that this solution is terser and therefore looks clearer than the previous solution – its Diffuseness is better. On further inspection we should also notice that we do not have to explicitly list the properties an expression depends on at the end. We only have to use an empty pair of parentheses when we write down a reactive property in an expression. If we forget the parentheses the compiler will complain (almost always) as, e.g., the type of `startDate.text` is different from and incompatible with the type of `startDate.text()`. So in addition the new solution's Viscosity is better due to less repetition[8] and the dependencies of an expression are still clear – they can be found right inside the expression where they should be. If anything the Abstraction Level is improved as one does not have to use a special `createXBinding` construct for the return type `X` of an expression. As is the case for local variables the return type is inferred with `|=`. All in all the new solution is a clear improvement that does not even have a negative trade-off for our dimensions.

For Timer (3.6) our remaining issue basically was the question if a timing construct that is better suited for changing requirements could be employed. The following is the relevant code excerpt.

```
val elapsed = DoubleProperty(0)
progress.progress <== elapsed / slider.value
numericProgress.text <== Bindings.createStringBinding(
    () => formatElapsed(elapsed())
  , elapsed)
reset.onAction = (event: ActionEvent) =>  elapsed() = 0

val timeline = Timeline(KeyFrame(Duration(100), "", (e: ActionEvent) =>
  if (elapsed() < slider.value()) elapsed() = elapsed() + 1))
timeline.setCycleCount(Timeline.INDEFINITE)
timeline.play()
```

And here is the solution with Scala.Rx.

[8] There is a project called EasyBind for JavaFX that also gets rid of the repetition but does not achieve the full notational convenience of Scala.Rx's approach. https://github.com/TomasMikula/EasyBind

```
val elapsed = Var(0)
progress.progress |= elapsed() / slider.value()
numericProgress.text |= formatElapsed(elapsed())
reset.onAction = (event: ActionEvent) =>  elapsed() = 0

val t = Timer(100 millis)
Obs(t) { Platform.runLater(if (elapsed() < slider.value()) elapsed() += 1) }
```

We observe again that the new solution is terser. In particular, the part concerning the time line definition. The new solution is a bit more general, too. For example, there is no need for a `KeyFrame` which improves the Abstraction Level. The Viscosity is improved as well since we would simply need to switch the `t` in `Obs(t)` with something that represents another external event. There is one problem with the new solution though. The single-threaded nature of JavaFX shines through due to the need to use `Platform.runLater`. Still, altogether this new solution is a slight improvement.

For Temperature Converter (3.4) we wondered if we might make do without a special bidirectional binding construct without losing clarity. Again, here is the relevant code excerpt.

```
celsius.text.bindBidirectional[String](fahrenheit.text, new StringConverter[String] {
  override def fromString(c: String): String =
    if (isNumeric(c)) cToF(c) else fahrenheit.text()
  override def toString(f: String): String =
    if (isNumeric(f)) fToC(f) else celsius.text()
})
```

And here is the solution with Scala.Rx.

```
celsius.text |= {
  if (isNumeric(fahrenheit.text())) fToC(fahrenheit.text()) else celsius.text.now
}
fahrenheit.text |= {
  if (isNumeric(celsius.text())) cToF(celsius.text()) else fahrenheit.text.now
}
```

Like with the other new solutions this one is terser. Its Abstraction Level is better as there is no need for a special bidirectional binding construct. The one subtelty to be aware of is to not create a self-referential dependency. This is why inside the expression for `celsius.text`, `celsius.text.now` is used instead of `celsius.text()`. The call to `celsius.text.now` gives the text value without registering an implicit dependency. An ideal solution would take care of such feedback loops automatically as far as possible. Nevertheless, this new solution is a slight improvement.

For Circle Drawer (3.8) one opportunity for improvement we identified was a clearer way to express the dependency of the variable `hovered`.

```
var hovered: Circle = null
// ...
onMouseMoved = (e: MouseEvent) => {
  hovered = getNearestCircleAt(e.x, e.y)
  draw()
}
```

We remember that our ideal solution would look akin to the following.

```
hovered = { getNearestCircle(mouse.x, mouse.y) }
WhenChanged(hovered) { draw() }
```

The problem is where do we get `mouse.x` and `mouse.y` from? If `mouse` were a global property we could probably achieve something like the following with our Scala.Rx integration.

```
var hovered = Rx{ getNearestCircleAt(mouse.x().relTo(this), mouse.y().relTo(this)) }
Obs(hovered) { draw() }
```

We can help ourselves in this case though. The following is what we *concretely* achieve with out integration which, apart from the first two lines, is pretty similar to our ideal wish.

```
val mouse_x = Var(0.0); val mouse_y = Var(0.0)
onMouseMoved = (e: MouseEvent) => { mouse_x() = e.x; mouse_y() = e.y }

val hovered = Rx{ getNearestCircleAt(mouse_x(), mouse_y()) }
Obs(hovered) { draw() }
```

For Cells (3.9) we wondered if there was a better way to implement change propagation. Since Scala.Rx describes itself as a change propagation library we might assume that an improvement is easily achieved with our current integration. This is unfortunately not the case. The main problem is that an `Rx` signal expression is not updateable as opposed to a `Var`[9]. The following is an abbreviated sketch of the old solution's relevant parts of the class `Cell`.

```
case class Cell(row: Int, column: Int) extends Observable with Observer {
  private var f: Formula = Empty
  private var v: Double = 0

  def formula: Formula = f
  def formula_=(f: Formula) = {
    for (c <- references(formula)) c.deleteObserver(this)
    this.f = f
    for (c <- references(formula)) c.addObserver(this)
    value = evaluate(f)
```

[9] There are most probably more obstacles even if a signal expression was updatable.

```
  }

  def value: Double = v
  def value_=(w: Double) {
    if (!(v == w || v.isNaN && w.isNaN)) {
      v = w
      setChanged()
      notifyObservers()
    }
  }

  override def update(o: Observable, arg: scala.Any) = {
    value = evaluate(formula)
  }
}
```

The following is a hypothetical ideal solution to at least give an idea how such a solution might look like.

```
case class Cell(row: Int, column: Int) {
  private val f: Rx[Formula] = Rx{ Empty }

  def formula: Formula = f()
  def formula_=(g: Formula) = {
    f update { g }
  }

  def value: Double = f().value
}
```

We see that the code would be simplified as we could assume that the change propagation is taken care of by the library. There are still open issues with the above example though. The field `f` is a reactive `Formula`. When `f` is updated to the new formula `g` then what should happen is that the referenced cells of `g` are identified automatically such that `f`'s dependencies can be updated accordingly. How this might work with Scala.Rx is not clear. In addition, we assume that a formula has a `value` function that returns its value which is however "outside" of the `Rx` so to say. It is not clear how the computation of `value` can be cached then without doing it manually in `Formula`'s definition. Or maybe it would have been better to let `Cell` extend `Rx` but then it seems that essentially the same amount of work would have to be done as in our Observer based solution.

To conclude, a sibling of FRP like Scala.Rx can definitely improve the expression of functional dependencies in GUI programs compared to JavaFX's resp. ScalaFX's current approach. The key to this improvement is that dependencies are conveniently established inside a signal expression with `()` right where they are used and that defining the signal expression in the first place does involve almost no boilerplate. We reached limitations of Scala.Rx

and our integration for the case study Cells but, hypothetically, we saw that improvements could be achieved.

4.2 ReactFX

ReactFX [38] is a reactive programming library for JavaFX. In the terminology of [5] ReactFX is a Cousin of FRP whereas Scala.Rx is a Sibling. That means, ReactFX focuses on event streams and asynchrony and not on time-varying values[10]. In this section's exemplary programs we pretend to have a perfect integration of ScalaFX and ReactFX[11]. Tomas Mikula, the author of ReactFX, generously helped improve the integration and code examples in this section as well as giving advice on best practices with ReactFX.

The basic building block of ReactFX is an `EventStream`. An `EventStream` emits values (events) and can be subscribed to in order to get notified each time a value is emitted.

```
val eventStream: EventStream[T] = ...
eventStream.subscribe(event => println(event))
```

To make itself inter-operable with JavaFX, ReactFX provides adapter methods to translate JavaFX-specific concepts into event streams. For instance, obtaining an event stream of mouse clicks on a specific node is achieved as follows.

```
val clicks: EventStream[MouseEvent] =
        EventStreams.eventsOf(node, MouseEvent.MOUSE_CLICKED)
clicks.subscribe(clickEvent => println("Click!"))
```

Obtaining an event stream of text values whenever a user changes the text of a text input is achieved like so.

```
val textValues: EventStream[String] = EventStreams.valuesOf(textInput.text)
textValues.subscribe(newValue => println("New text: " + newValue))
```

An `EventStream` has a number of event combinators with which a new `EventStream` can be created with different behavior from the original event stream. For example, to retrieve an event stream of left mouse clicks we can employ the `filter` combinator.

```
val clicks: EventStream[MouseEvent] =
        EventStreams.eventsOf(node, MouseEvent.MOUSE_CLICKED)
val leftClicks: EventStream[MouseEvent] =
        clicks.filter(click => click.button == MouseButton.LEFT)
```

There are many more combinators part of which we will come to know in the rest of this section. We already mention here that most improvements using

[10] We remind ourselves that JavaFX itself already has time-varying values in the form of `ObservableValue` which in particular extends JavaFX properties.

[11] The examples in fact are implemented in Java/JavaFX as well as Scala/ScalaFX but they are more verbose. The issue with our Scala/ScalaFX integration is that the expressions involving ReactFX constructs need a lot of type hints to make the Scala compiler accept the programs. To not distract the analysis with these current unfortunate but also unfundamental shortcomings we assume a perfect integration.

an FRP cousin like ReactFX over plain JavaFX are achieved when the problem involves something that is naturally represented as events (e.g. mouse events or timers) and when clever stream compositions can be employed to express otherwise complicated (asynchronous) interactions succinctly.

Although ReactFX provides these aforementioned means of inter-operation with JavaFX we want to take its integration with ScalaFX further in order to have a comparable integration to the previous section and also to not distract the analysis with in principle unnecessary boilerplate code. As before we use Scala's Implicits. The following is a sketch of this integration.

```
object ReactFXIntegration {
  implicit def obsValToStream[T](p: ObservableValue[T]): EventStream[T] =
    EventStreams.valuesOf(p)

  implicit class PropertyExtensions[T](p: Property[T]) {
    def |=(s: EventStream[_ <: T]) = s.subscribe(v => p.value() = v)
  }

  implicit class NodeExtensions(n: Node) {
    def actions: EventStream[ActionEvent] =
      EventStreams.eventsOf(n, ActionEvent.ACTION)
    def mousePresses: EventStream[MouseEvent] =
      EventStreams.eventsOf(n, MouseEvent.MOUSE_PRESSED)
    // ...
  }
}
```

Using this integration we can obtain the previous event stream of text values without the need to introduce `textValues`.

```
textInput.text.subscribe(newValue => println("New text: " + newValue))
```

What happens behind the covers is that basically `textInput.text` is transparently wrapped due to the definition of `obsValToStream` like so.

```
EventStreams.valuesOf(textInput.text).subscribe(newValue => println(...))
```

The `|=` operator is analogous to the previous chapter. With it we can assign changes from the event stream on the right hand side to the respective property on the left hand side. So,

```
textInput.text |= anEventStream
```

is equivalent to

```
anEventStream.subscribe(eventValue => textInput.text() = eventValue)
```

Finally, we provide an event stream counterpart to JavaFX `onAction`, `mouseClicks` etc. event handlers on the class `Node`.

Now, let us explore ReactFX's applicability to our case studies. The following shows the relevant excerpt of the Counter (3.3) case study.

```
val count = new TextField { text = "0" }
val countUp = new Button("Count")

countUp.onAction = (event: ActionEvent) => {
  count.text = (1 + count.text().toInt).toString
}
```

And here is a naive approach using ReactFX.

```
val count = new TextField { text = "0" }
val countUp = new Button("Count")

countUp.mouseClicks.subscribe(click =>
        count.text = (1 + count.text().toInt).toString)
```

Conceptually, both excerpts are equivalent. In both we are using the side-effect of explicitly setting the text value of `count` and we have inversion of control. We can however reduce the "side-effectfulness" with ReactFX and do without inversion of control, thus making the solution more functional and clearer.

```
val count = new TextField { text = "0" }
val countUp = new Button("Count")

count.text |= countUp.actions.accumulate(0, (n, _) => n + 1)
```

The stream combinator `accumulate` creates a new stream that applies a transition function on a state value whenever a new event from the original stream is emitted. This state value is given an initial value. Here, the transition function is incrementation by one[12] with the initial value zero and the original event stream are the clicks on the button `countUp`. That is, we have a stream that counts the button clicks – exactly what we want. And by binding the values of this accumulated stream to `count`'s text with the `|=` operator we display the count. Having seen both a naive use of ReactFX and a more reasonable, we take the opportunity to formulate an advice for profitably using ReactFX. One should try to minimize the use of `subscribe` as this function is inherently side-effectful. Ideally, `subscribe` should only be used at the boundary between ReactFX and non-ReactFX code.

The Abstraction Level of the new solution is slightly higher as we need to be aware of the `accumulate` combinator and know that it can be used in this situation. However, these combinators are so fundamental to ReactFX and allow to express more complicated abstract concepts more easily that this is really a minor point. The Closeness of Mapping is better without the inversion of control of callbacks. Due to doing without a callback this new solution also has no Hidden Dependencies which are otherwise contained in the body of a callback. The new solution has less Error-proneness and Diffuseness

[12] The underscore in the code is simply the button click event value which we ignore.

because of the lack of integer and string conversions and assignment. All in all, ReactFX's solution is an improvement over the original solution.

Next in turn is Temperature Converter (3.4). Again, here is the relevant original code excerpt.

```
celsius.text.bindBidirectional[String](fahrenheit.text, new StringConverter[String] {
  override def fromString(c: String): String =
    if (isNumeric(c)) cToF(c) else fahrenheit.text()
  override def toString(f: String): String =
    if (isNumeric(f)) fToC(f) else celsius.text()
})
```

And here is the solution with ReactFX.

```
celsius.text |= fahrenheit.text.filter(v => isNumeric(v)).map(v => fToC(v))
fahrenheit.text |= celsius.text.filter(v => isNumeric(v)).map(v => cToF(v))
```

On the right hand side of the first line we first have the stream `fahrenheit.text` that emits a new event containing the current text in the input field `fahrenheit` whenever the user changes the text in said input field. With the `filter` combinator we create a new stream that only retains event values that constitute a number. In particular, this means that when the user enters a malformed string the filtered stream will not emit a new event. Finally, we create yet another stream that maps the filtered Fahrenheit event values into Celsius event values. This final stream is then bound to the text property of the input field `celsius`. The explanation for the second line is dual. In this case, feedback loops are prevented automatically[13].

The new solution is more succinct so the Diffuseness is better. On the one hand, the Abstraction Level is better because we do not need a special toolkit-provided bidirectional binding construct but instead can employ basic building blocks to achieve the same effect. On the other hand, the Abstraction Level is somewhat worse compared to the solution in Scala.Rx because we need to express the right hand side with ReactFX's combinator sub-language and cannot reuse the host language's native control flow constructs. Then *again*, the combinators `filter` and `map` are so basic to ReactFX that we still consider the Abstraction Level as very good. Error-proneness is reduced as feedback loops are automatically taken care of. The other dimensions are more or less the same for both solutions so we conclude that we have another improvement with ReactFX.

Let us now consider Flight Booker (3.5). We base our comparison on our result with Scala.Rx from the previous section.

[13] Feedback loops do not happen due to the semantics of `EventStreams.valuesOf(...)`. Basically, `valuesOf` creates a new stream from its input observable value i such that for a succession of identical emitted values from i only the first is retained and the rest is ignored until a first non-duplicate value is emitted again by i. For an explanation how to prevent feedback loops in a situation where the conversion back and forth is not stable (i.e. `fToC(cToF(v)) != v`) see appendix C.

```
returnDate.disable |= flightType.value() == "one-way flight"
startDate.style  |= { if (isDateString(startDate.text()))  "" else "red" }
returnDate.style |= { if (isDateString(returnDate.text())) "" else "red" }
book.disable |= {
  flightType.value() match {
    case "one-way flight" => !isDateString(startDate.text())
    case "return flight" =>
      !isDateString(startDate.text()) || !isDateString(returnDate.text()) ||
      stringToDate(startDate.text()).compareTo(stringToDate(returnDate.text())) > 0
  }
```

The result with ReactFX is as follows.

```
returnDate.disable |= flightType.value ==== "one-way flight"
startDate.style  |= startDate.text.map(t => if(isDateString(t)) "" else "red")
returnDate.style |= returnDate.text.map(t => if(isDateString(t)) "" else "red")
book.disable |= combine(flightType.value, startDate.text, returnDate.text)
    .map { case (ft, sd, rd) => ft.match {
      case "one-way flight" => !isDateString(sd)
      case "return flight" =>
          !isDateString(sd) || !isDateString(rd) ||
          stringToDate(sd).compareTo(stringToDate(rd)) > 0
    }}
```

The high-level structure of this solution is similar to the solution in Scala.Rx and, likewise, to the solution in pure ScalaFX. The differences to the solution in Scala.Rx are the following. In the first line me must use a special equals operator[14] defined for event streams. In the next two lines we use the `map` combinator to transform a date string into a color depending on the validity of the date string. Finally, we use the `combine` combinator together with the `map` combinator to describe the functional dependency of the book button. The `combine` combinator takes as input several event streams and creates an event stream that emits a tuple of the events of its inputs whenever one of its inputs emits.

As the high-level structure is similar the Closeness of Mapping is similar as well. The new solution's Diffuseness, too, is similar to the solution in Scala.Rx. The Abstraction Level and the Viscosity are arguably worse because we have the same problem as with the `createXBinding` solution in pure ScalaFX (or pure JavaFX in that respect): we need to explicitly list the dependencies of the binding expression. However, compared to the solution in pure ScalaFX/JavaFX the Error-Proneness is reduced since the compiler will complain if we, for example, add another dependency as an argument to `combine` but forget to adapt the first `case` statement. We cannot almost completely express the bindings using the host language's native control flow constructs as it is the case for Scala.Rx. We need to express parts of the bindings in terms of ReactFX's combinators but this is a general trade-off of

[14] We have it defined in the ReactFX integration but did show its definition in this section. It is a relatively easy addition to `PropertyExtensions`.

using ReactFX. To make a long story short, although the new solution is by no means bad we do not gain as much with ReactFX over pure ScalaFX as we did with Scala.Rx[15].

The original excerpt from Timer (3.6) is as follows.

```
val elapsed = DoubleProperty(0)
progress.progress <== elapsed / slider.value
numericProgress.text <== Bindings.createStringBinding(
    () => formatElapsed(elapsed())
  , elapsed)
reset.onAction = (event: ActionEvent) =>  elapsed() = 0

val timeline = Timeline(KeyFrame(Duration(100), "", (e: ActionEvent) =>
  if (elapsed() < slider.value()) elapsed() = elapsed() + 1))
timeline.setCycleCount(Timeline.INDEFINITE)
timeline.play()
```

The result with ReactFX is as follows.

```
val resets = reset.actions
val ticks = EventStreams.ticks(Duration.ofMillis(100))
val elapsed =
      StateMachine.init((0.0, slider.value))
        .on(resets).transition{case ((_, s), _) => (0.0, s)}
        .on(ticks).transition{case ((e, s), _) => (e + (if (e < s) 1 else 0), s)}
        .on(slider.value).transition{case ((e, _), s1) => (e, s1.doubleValue)}
        .toStateStream.map{case (e, _) => e}

progress.progress    |= combine(elapsed, slider.value).map((e, s) => e / s.doubleValue)
numericProgress.text |= elapsed.map(e => formatElapsed(e))
```

At first, two event streams are given a name, namely `resets` and `ticks`. The stream `resets` represents the clicks on the button `reset` and the stream `ticks` simply emits a new event every 100 milliseconds. Then, the stream `elapsed` is defined as a state machine whose state is a tuple of two `Double` values where the first component stands for the elapsed time e from the case study's description and the second component simply "drags along" the slider value. The general stream combinator `StateMachine` from ReactFX allows us to describe a stream which has a state value that can be transformed by declaring transition functions that depend on other event streams. Observe that the `accumulate` combinator from before is a special case of `StateMachine` with exactly one transition function. In our case, we have three transition functions. The first resets e to zero whenever `reset` is clicked[16]. The second increments e by one as long as e has not reached the

[15] Tomas Mikula, the author of ReactFX, in fact does not recommend employing ReactFX for such a use case. We simply presented this solution to not only understand the strengths of an approach but also the use cases where a particular approach would not be as beneficial as other alternatives.

[16] The first underscore in `case ((_, s), _)` represents e and the second underscore is the (button click) event which we both ignore.

slider value. And the third simply updates the slider value in the state tuple whenever the user moves the slider. Note that we use Scala's case syntax for anonymous functions that allows us to destructure the tuple on the left side of the anonymous functions[17]. In the last two lines, we bind the streams to their respective properties in order to display the stream values.

The Abstraction Level is arguably better as we can describe the behavior of the application solely with stream combinators without making the solution significantly harder to understand. In the original solution the time-dependent behavior is achieved with the *special* time line construct of JavaFX (which is by no means a bad way to model) and a callback whereas with ReactFX we take a more consistent approach with a tick stream and a state machine stream combinator that is more general yet not much harder to understand than the time line construct. Doing away with callbacks in the traditional sense and instead using transition functions of the state machine combinator forces us to be more explicit with regard to the Hidden Dependencies. Having to use `combine` to bind the ratio to `progress.progress` leads to more Diffuseness and has arguably a worse Closeness of Mapping[18]. Apart from that, the Closeness of Mapping is more or less equally good even though the solutions each take a different conceptual approach. Needing to drag the slider value in the state machine's state along increases the Diffuseness but also the Viscosity as adding another tuple component to the state would lead to adapting all the transition functions. It is more convenient to simply refer to outer variables inside a callback. However, the explicit "no side-effect" approach improves the dimensions Hidden Dependencies and Error-proneness. To sum up, the new solution is definitely more functional and can also be considered an improvement over the original solution. How much of an improvement it is considered depends on which dimensions one places the most value.

The following is the relevant excerpt from the original solution for Circle Drawer (3.8).

```
var hovered: Circle = null

val diameter = new Button("Diameter...")
val popup = new Popup()
popup.content.add(diameter)

diameter.onAction = (e: ActionEvent) => {
  popup.hide()
  showDialog(hovered)
```

[17] Without it, e.g., the first anonymous function would be written as `(tup_es, _) => (0.0, tup_es._2)`.

[18] In principle, we could have extended our integration to make it possible to write `progress.progress |= elapsed / slider.value` so we will not judge this point severly. In practice, it was not managed but a sketch of such an integration is still provided in code listing 17 in appendix F.

```
}
onMousePressed = (e: MouseEvent) => {
  if (e.isPrimaryButtonDown && hovered == null) {
    addCircle(new Circle(e.x, e.y))
    onMouseMoved.get.handle(e)
  }
  if (popup.isShowing) popup.hide()
  if (e.isPopupTrigger && hovered != null)
    popup.show(this, e.screenX, e.screenY)
}
onMouseMoved = (e: MouseEvent) => {
  hovered = getNearestCircleAt(e.x, e.y)
  draw()
}
```

And here is the new solution with ReactFX.

```
val diameter = new Button("Diameter...")
val popup = new Popup()
popup.content.add(diameter)

val leftPressesToVoid = this.mousePresses
        .filter(e => e.isPrimaryButtonDown)
        .filter(e => getNearestCircleAt(e.x, e.y) == null)
val addedCircles = leftPressesToVoid
        .map(e => new CircleFX(e.x, e.y))
        .hook(c => addCircle(c))
val hoveredCircles = this.mouseMoves
        .map(e => getNearestCircleAt(e.x, e.y))
merge(addedCircles, hoveredCircles).subscribe(c => draw(c))

this.mousePresses.subscribe(e => if (popup.isShowing) popup.hide())
val rightPressesToCircle = this.mousePresses
        .filter(e => e.isPopupTrigger)
        .filter(e => getNearestCircleAt(e.x, e.y) != null)
rightPressesToCircle.subscribe(e => popup.show(this, e.screenX, e.screenY))
val selectedCircles = rightPressesToCircle
        .map(e => getNearestCircleAt(e.x, e.y))
selectedCircles.emitOn(diameter.actions).subscribe(c => {
          popup.hide()
          showDialog(c)
        })
```

The first three lines initialize the structure of the right-click menu. The next section defines the behavior of adding and hovering over circles and the final section defines the behavior of the pop-up menu. The first stream `leftPressesToVoid` represents the events of left-clicking on empty space[19]. The stream `addedCircles` contains as events the created circles. We remember that a circle is created whenever the user left-clicks on empty space. The `hook` combinator introduces a side-effect into a stream and returns it

[19] We remember that `getNearestCircleAt` returns `null` if a mouse click is outside every circle.

unchanged[20]. The side-effect is adding a new circle to our list of circles in order to display it and have it considered in the snapshot history. We note that the `draw` method now takes the circle to shade gray as an argument[21]. Since we want to have a circle shaded gray when the mouse hovers over it we need the stream `hoveredCircles`. If the mouse does not hover over any circle `hoveredCircles` emits `null`. As the final step in this section we need to redraw the screen when adding or hovering over new circles which is achieved with the `merge` combinator that emits an event whenever either of its input streams emits. The next section begins with a simple callback that hides the pop-up on any click if it is visible. The stream `rightPressesToCircle` is very similar to `leftPressesToVoid` with the difference that it emits right-clicks and only those that happened inside a circle. In such a case we want to show the pop-up menu which is achieved with the next line. Now, we need to remember the right-clicked or rather selected circle to provide it as an argument to `showDialog`[22]. We achieve this with the last two definitions. The stream `selectedCircles` emits the right-clicked circle. If we tried to write something along

```
diameter.actions.subscribe(e => { popup.hide(); showDialog(???) })
```

in order to create the corresponding dialog for the selected circle we would have no way of providing `showDialog` with its necessary argument (the selected circle). This is why we use the `emitOn` combinator. The stream `selectedCircles.emitOn(diameter.actions)` emits the latest event from `selectedCircles` whenever (and only then) `diameter.actions` emits. The argument to `emitOn` acts as an impulse so to say.

On the one hand, the Abstraction Level is better because the new solution is more consistent. It uses streams throughout, does away with the arguably unfunctional `hovered` field and clearly divides side-effecting operations from side-effect free ones. On the other hand, we need to employ relatively many different combinators whereas the original solution got along with only callbacks. That said, the new solution might only seem more complex due to unfamiliarity with such a new paradigm. The aforementioned division of operations and ability to give intermediate streams meaningful names improves the Closeness of Mapping to the desired behavior of the case study in this case. The new code has more Diffuseness but not severely so[23]. The new solution eliminates the `hovered` field and the function `draw` now takes the hovered circle as an argument. This design is more functional and it

[20] One could have also used `subscribe` in this case (`addedCircles.subscribe(...)`). Using `hook` in general is more efficient though as it is lazy, that is, the side-effect is only performed if the stream as at least one subscriber.

[21] If the argument is `null` then simply no circle is shaded.

[22] "Dialog Control". In the original solution we used the variable `hovered` in this situation.

[23] Code Listing 18 in appendix F shows a more compact way to write the final section. That is, the Diffuseness with ReactFX can actually be better but we kept the slightly more verbose version here in order to make the explanation clearer.

improves the dimensions Hidden Dependencies and Error-proneness. The dependency on the hovered circle of the function `draw` is now explicit and without `hovered` as a field that is assigned and read inside callbacks (which is a side-effect and always somewhat implicit) the streams now must make it explicit when they depend on a hovered circle. Error-proneness is reduced because the possibility of `hovered` being changed outside the callbacks is eliminated. There are still side-effecting operations in the new solution but these are clearly separated and essential. Concluding, this alternative approach can definitely be considered an improvement over the old one.

For the CRUD (3.7) and CELLS (3.9) case study we did not find a reasonable way to work ReactFX in[24].

The following example is taken from the author of ReactFX Tomas Mikula's blog[25] in order to demonstrate a use case where a lot of advantages can be achieved with a cousin of FRP but which we did not cover in our case studies. The task is syntax highlighting. When the user pauses typing for 500 milliseconds a background computation for syntax highlighting is triggered. When this background computation is completed, the resulting highlighting is applied to the text but *only* if the text has not changed in the meantime. From this task description we can clearly identify the requirements of asynchrony and constraints on this asynchrony[26]. Let us see how ReactFX solves this task. However, before doing so we need to introduce some assumptions. The following definitions relate to the syntax highlighting.

```
// Represents highlighting.
class Highlighting { ... }
// Computes highlighting for the given text.
def computeHighlighting(text: String): Highlighting = ...
// Applies the given highlighting to the text area.
def applyHighlighting(highlighting: Hightlighting) = ...
// Computes highlighting in the background.
def computeHighlightingAsync(text: String): Task[Highlighting] = ...
```

The solution then looks as follows.

```
1 val textValues = EventStreams.valuesOf(textArea.text)
2 val cancelImpulse = textValues
3 textValues.successionEnds(Duration.ofMillis(500))
```

[24] If asynchrony were a challenge in CRUD then ReactFX could have been used to create a stream for each of the C, U, D requests, start an asynchronous task (e.g. a REST HTTP request) for each, await the completion or report an error.

[25] http://tomasmikula.github.io/blog/2014/04/25/combining-reactfx-and-asynchronous-processing.html

[26] It could be held against our case studies that none of them really tests such an asynchronous behavior which is a fair criticism. As future work one of the case studies could be replaced (for example TIMER) with a case study similar to this example or the current set of case studies could simply be extended.

```
4       .mapToTask(this::computeHighlightingAsync)
5       .awaitLatest(cancelImpulse)
6       .subscribe(this::applyHighlighting)
```

Line 1 creates an event stream that emits the new value of `textArea`'s text every time its text changes. Line 2 just gives another name to the same stream. We are going to use every new value of text to cancel the asynchronous computation currently in progress. On line 3 we only keep the text values that have not changed in the last 500 milliseconds. Line 4 starts the asynchronous computation of highlighting for text values that survived the filter from line 3.

Line 5 creates a stream that emits the results of the asynchronous computations started in line 4. It only emits those that have not been outdated by the time they are completed. An asynchronous computation is considered outdated if either another asynchronous computation starts before it is completed, or an impulse to cancel the computation arrives[27]. As a consequence, a highlighting that is emitted from the stream on line 5 is up to date and can be applied to the text in `textArea`. Line 6 just applies every highlighting emitted on line 5 to the text in `textArea`.

To make the asynchronous nature this task clearer we provide a diagram[28] in figure 4.1 where the x-axis represents the time and the y-axis represents the streams. Circles represent events. In the depicted situation the user submits three key strokes. The second comes 200ms after the first and the third comes 600ms after the second. The second event cancels the first because it happened before 500ms passed. The third event cancels the second because it emits a cancel impulse while the background task computes the highlighting. The third one comes through.

Solving this task without the help of ReactFX in pure JavaFX would be *much* more involved. The solution as it is presented here is a very succinct and clear way to solve the syntax highlighting problem. In particular, we see that the real convenience of ReactFX comes into play when ReactFX's numerous combinators can be employed in a fruitful way for complicated asynchronous requirements.

In conclusion, a cousin of FRP like ReactFX is beneficial when the problem involves something that is naturally represented as events (e.g. mouse clicks or timers). In these situations, using ReactFX leads to more functional solutions which have less hidden dependencies and are, in general, less error-prone due to eliminating or at least separating side-effect operations from compositional stream operations. Most benefits can be achieved for problems involving asynchrony where clever stream compositions can be employed to

[27] In this case, the first situation will not happen because `textValues` and `cancelImpulse` are the same stream.

[28] This diagram is somewhat inspired by Marble Diagrams from .NET Rx.

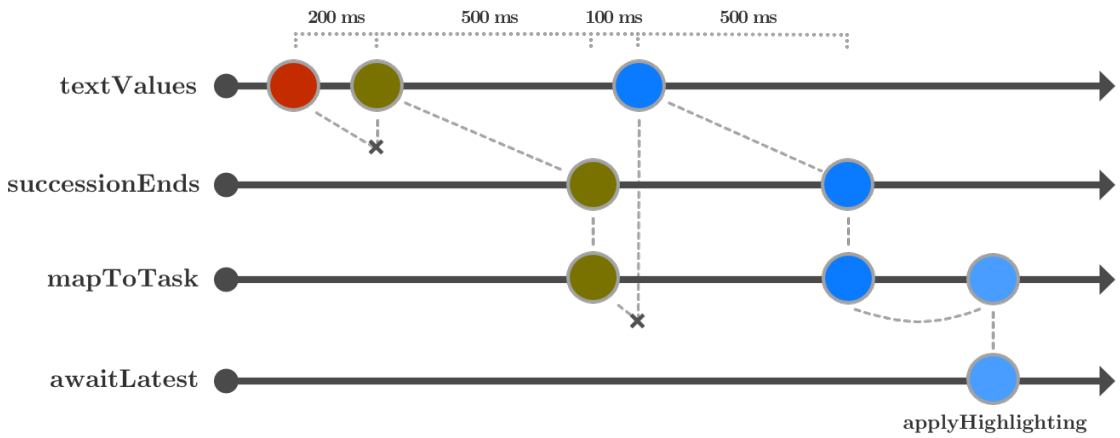


Figure 4.1: A depiction of situations for an asynchronous solution with ReactFX for syntax highlighting.

express otherwise complicated interactions succinctly. Expressing functional dependencies is, however, not a particular strength of a Cousin of FRP.

Such an approach to GUI programming as presented in this section is actually already used profitably in practice. The most prominent example is perhaps the Github Mac App[29] which is entirely written in this style using the ReactiveCocoa[30] framework. Another framework with a somewhat similar approach is ReactiveUI for .NET[31].

[29] https://mac.github.com/

[30] https://github.com/ReactiveCocoa/ReactiveCocoa and https://github.com/blog/1107-reactivecocoa-for-a-better-world

[31] http://www.reactiveui.net/

4.3 Elm

A lot of GUI libraries even in pure functional languages are essentially a thin shim over an (imperative) object-oriented GUI toolkit. This means, for example, that widgets are not pure functions. Widgets have state and therefore an identity. If we metaphorically turn up the functional knob on a GUI toolkit's control board to its maximal value we must arrive at a situation where even the widgets are nothing but pure functions. Elm is such an example of a maximally functional language and GUI toolkit at the same time. As callbacks are all about side-effects Elm naturally does not use callbacks whatsoever for behavior-related code but instead FRP. Where customization in an OOP language is most often achieved with inheritance, in Elm there is only the way of function composition[32]. In short, it does not get any more functional than using Elm for writing GUIs. Among other aspects, we will investigate if this "radical" approach has benefits or not[33].

To provide a gentle introduction to the language, we present a very simple program[34].

```
import Mouse
main = lift asText Mouse.position
```

The program displays the position of the mouse as it moves around the screen, automatically updating the screen. The value of the variable `main` is by convention displayed on screen. The function `asText` turns any value into a displayable textual representation. The *signal* `Mouse.position` is a value that changes over time[35]. Now, the function `lift` is needed to "lift" or transform the function `asText` from one that works on values to a function that works on signals (as `Mouse.position` is a signal and *not* a conventional value). In this way, the function `lift` ensures that `asText` is applied to `Mouse.position` every time the mouse moves.

We see that Elm's syntax and especially its function application syntax is different from the examined languages so far. To make it clearer how to read this example we give an analogous pseudo "Elm-in-Scala" snippet.

```
import Mouse
object HelloWorld extends JFXApp {
    stage = lift(asTextLabel)(Mouse.position)
    // or conceivably
    stage = lift(asTextLabel, Mouse.position)
    // or with implicit conversions presumably
    stage = Mouse.position.asTextLabel
}
```

[32] Even in OOP composition is a best practice and the use of inheritance should always be critically weighed against alternatives.

[33] The beginning of this section has been sanity checked by Evan Czaplicki, the author of Elm.

[34] The program and the explanation is based on Elm's Wikipedia article and [13].

[35] Analogous to JavaFX's `ObservableValue`.

All other possibly unfamiliar constructs will be explained where appropriate when we analyze Elm's solutions for our case studies.

To refer back to the classification of FRP systems from the introduction, in a sense, Elm is both a Sibling and a Cousin of FRP. Signals in Elm combine both the time-varying and change propagation aspect of `Rx` expressions from Scala.Rx and the event stream concept from ReactFX[36]. Elm also provides constructs for asynchrony.

As always, let us begin with the Counter (3.3) case study. A solution in Elm is as follows.

```
clicks = input 0

display count =
    let l = label (show count)
        b = button clicks.handle (count + 1) "Count"
    in hbox [l, b]

main = lift (frame "Counter" . display) clicks.signal
```

An Elm program can be thought of as a signal graph whose nodes are connected by directed edges that indicate the flow of values. In the beginning of this section we have seen the signal `Mouse.position` which is an input node in such a signal graph, i.e. a node with no incoming edges and one or more outgoing edges, that is provided by the Elm system. With `clicks = input 0` we define our own input signal whose value/state is the number of clicks issued by the user. More precisely, we not only get a signal `clicks.signal` but also a *handle* `clicks.handle` by which we can refer to this input node. Next, we define function `display` that takes as its argument a number `count` and creates both a label `l` showing the count and a button `b` to increase the count. With a *let* expression we can give names to expressions and refer to them in the body of the let expression. The function `label` expects a string which is why `count`, a number, is transformed to a string with the function `show`. The function `button` takes a handle, a new value to update the node's value and a string to label the button. In our case we set the value stored in `clicks` to `count + 1`. The function `hbox` is a layout combinator inspired by JavaFX's HBox layout that takes a list of elements[37] and aligns them horizontally. Finally, we bring it all together in the expression on the right-hand side of the variable `main`. The function `frame` creates a titled frame containing the given element as the second

[36] In earlier FRP languages these two concepts were separate. The first concept was commonly called *behavior* and the second *event* or *event stream*. In ReactFX, an event stream does not keep its last value. In Elm, signals always have a value (akin to behaviors) and if we have something like an ReactFX event stream in Elm as a signal, the signal always has the latest emitted value from this stream.

[37] The square brackets denote a list.

argument. Now, we cannot give `display` itself as the second argument to `frame` because `display` is a function and not an element. So we need to apply `display` to its argument to get the desired element. However, its argument is a signal so we need to lift `display`. As we then would need to lift `frame` in turn we composed both functions with the dot operator to shorten the code. The following definition of `main` employing a lambda expression would be equivalent.

```
main = lift (\count -> frame "Counter" (display count)) clicks.signal
```

To sum up, whenever the user clicks the button `b`, `b` issues an update to `clicks.signal`. It can do this because it was given a handle to `clicks`. Then, the expression on the right-hand side of `main` is reevaluated and `display` receives the current value of `clicks.signal` which is bound to `display`'s parameter `count`. Due to `b`'s definition the value of `clicks` is increased by one. Lastly, the screen is redrawn. This is how we achieve the functionality of a counter as described in the case study.

As was the case with the first solution in ReactFX for the Counter case study the current solution in Elm can be improved. A best practice in Elm is to view the state of a program as a centralized store which makes it easier to guarantee consistency across many different views and components. And views in turn should only know as much of the state as absolutely needed to present themselves. With this in mind a more idiomatic solution would be if the view function (and therefore the button) would really only draw its content without knowing that the count must be increased. We can push the knowledge about incrementation into the signal instead.

```
clicks = input ()

display count =
    let l = label (show count)
        b = button clicks.handle () "Count"
    in hbox [l, b]

main = lift (frame "Counter" . display) (count clicks.signal)

-- or equivalently using the operator <~
main = frame "Counter" . display <~ count clicks.signal
```

This improved solution is now somewhat similar to our solution with ReactFX. Since we do not care anymore about the value of the clicks input we give it the canonical empty value `()`. The function `count` takes a signal i as an argument and creates a new transformed signal c that represents the number of times i has emitted an event, i.e. c is the count of i. In our solution with ReactFX the stream combinator `accumulate` had the same role as `count`[38].

[38] As a remark, `count` is a special case of the `foldp` function that will be introduced later in this section.

Note that often parentheses can be avoided by using the operator `<~` instead of the function `lift`.

Compared to our previous solutions we note that there is almost no ceremony to create something that is displayed on the screen which reflects positively on the Diffuseness. The need for explicit lifting, however, increases the Diffuseness and Viscosity of the solution. For an example of increased Viscosity, suppose we had a function that worked on signals. Later, we decide that it could just as well work on normal values and change the function signature accordingly. Then we would need to adapt all the call sites of said function because it has to be lifted. Explicit lifting is fundamental to Elm though so that it stays pure and efficient. It is therefore a non-negotiable trade-off. As such, we won't repeat this general observation in the following analyses. As we have seen, there is a hypothetical potential to introduce Hidden Dependencies when using `input`. Apart from that everything else is very explicit. Due to Elm's natural way to describe hierarchical layout expressions and the ability to directly think of the application's behavior as a signal graph here and in general Elm has a good Closeness of Mapping to the resulting GUI applications. On the other hand, we cannot think, for example, of `b` as an object with an identity that we can refer to later when we want to e.g. change its padding[39]. The button `b` must rather be thought of as more or less a function that merely draws a button[40]. Elm's Abstraction Level is relatively steep especially for potential users who are unfamiliar with pure functional programming. Elm insists on the aforementioned separation of signals from conventional values which has extensive implications on how code must be structured and constrains it in a way that OOP languages do not. For example, we cannot simply write the following.

```
display =
    let c = count clicks.signal
        l = label (show c)
        b = button clicks.handle c "Count"
    in hbox [l, b]
```

Instead, if we wanted to create an OOP-like display component with encapsulated state we could write something as follows.

```
display phantom =
    let clicks = input 0
        c = count clicks.signal
        l = label . show <~ c
```

[39] This way of thinking about widgets is touted as a natural way of thinking by OOP languages and, indeed, it is generally accepted as such which is why we assign this approach a good Closeness of Mapping.

[40] From this perspective, Elm has more in common with *immediate-mode* GUI libraries (the drawing of the circles in Circle Drawer is an example for this approach) than with *retained-mode* GUI libraries (JavaFX except its canvas object is an example for this approach due to its scene graph).

```
        b = lift (\c -> button clicks.handle c "Count") c
    in hbox <~ combine [l, b]

main = frame "Counter" . hbox <~ combine [display (), display ()]
```

To show that the state really is encapsulated we created two counts that can be incremented independently. The parameter `phantom` is needed to prevent the Elm compiler to optimize the function `display` into a variable[41]. The combinator `combine` creates a 'signal of a list' from a 'list of signals'. We observe that the resulting code looks rather noisy which is a indication that this kind of code structure that is more common in OOP is highly discouraged in Elm. Concluding, there are a lot of new concepts to learn in Elm to make use of it, Elm insists on the syntactical separation and encourages the architectural separation of signal-related code from other code as well as the need to use special primitives (here `combine`) that work solely on signals. Encapsulation *of state* as in OOP is not encouraged in Elm[42]. Elm forces us to be explicit about state and encourages us to divide data, display as well as user interaction code and as such reduces Hidden Dependencies and possibly Error-Proneness. The Abstraction Level is steep and strict but the promise of Elm is that the constraints and conceptual costs for these abstractions will pay off for more complex tasks.

In this basic introductory case study we did not really see which practical pros and cons Elm's approach has for GUI programming but we at least had a glimpse at Elm's unique and novel nature. In the following analyses, we will learn more about the practical consequences.

Our next case study to be analyzed is Temperature Converter (3.4). The solution in Elm is as follows[43].

```
-- Model

convert : (Float -> Float) -> Content -> Content -> Content
convert f field current =
    let round' x = toFloat (round (x * 10)) / 10 in
    case String.toFloat field.string of
      Just n -> { field | string <- show (round' (f n)) }
      Nothing -> current

celsius : Input Content
celsius = input Field.noContent
fahrenheit : Input Content
fahrenheit = input Field.noContent
```

[41] Otherwise, both `display`'s would share the same count.

[42] On the other hand, using modules as a means of information hiding is highly encouraged in Elm. See Appendix D for an example of a typical module structure in Elm. Encapsulation is only discouraged in the sense that these modules should not hold state.

[43] The solution presented here was created by the author of Elm Evan Czaplicki after a discussion on Elm's mailing list with minor changes by the author of this thesis.

```
temperature : Signal Content -> (Float -> Float) -> Signal Content -> Signal Content
temperature normal f other =
    let conversions : Signal (Content -> Content)
        conversions = merge (always <~ normal) (convert f <~ other)
    in foldp (<|) Field.noContent conversions
    -- (<|) <~ conversions ~ normal -- This would not be enough.

-- View

display cHandle fHandle c f =
  hbox
  [ field cHandle id "Celsius" c
  , label "Celsius ="
  , field fHandle id "Fahrenheit" f
  , label "Fahrenheit"
  ]

main =
  let fields = display celsius.handle fahrenheit.handle
                       <~ temperature celsius.signal fToC fahrenheit.signal
                        ~ temperature fahrenheit.signal cToF celsius.signal
  in frame "Temperature Converter" <~ fields
```

The first thing that might stand out is that we present type information for some of the variables and functions to make their purpose clearer. Like in Scala the type comes after the colon – this is were the similarities end, though. Without going into too much detail the type of `convert` can be read as follows: `convert` takes as parameters a function `f` from floats to floats (`Float -> Float`), a value `field` of type `Content` and another value `current` of the same type and returns a new value of type `Content`. The type `Content` represents the content of a text field and contains not only a text string but also a cursor position[44]. The variables `celsius` and `fahrenheit` are analogous to `clicks` from the previous case study with the difference that they do not keep an integer as their state but instead a value of type `Content`, exactly what we need for a text input. `Field.noContent` is just the empty value for a `Content`.

The function `convert` is a little intricate. Therefore, we explain it in concrete terms to make it more approachable. Let us suppose that `f` is `fToC`, `field` is `fahrenheit`'s content and `current` is `celsius`' content. Now, if the content of `fahrenheit` is numeric (this is the `Just` case) then the result is a new `Content` whose string value is the respective temperature in Celsius[45].

[44] The reason for needing to consider the cursor position is somewhat subtle. Suffice it to say that in an OOP language the cursor position would probably be encapsulated state inside the text widget/object. As in Elm widgets have no identity/state this information has to be kept externally.

[45] This interesting syntax with the braces and `field` on the left means: create a new record whose contents are exactly like `field`'s except for the changes indicated on the right hand side of the vertical bar. Therefore, the cursor position is adopted from `field`.

Otherwise, `celsius`' content is returned as is. This is needed to have the behavior as demanded in the case study. That is, "when the user enters a non-numerical string into T_C the value in T_F is *not* updated and vice versa".

The heart of this implementation lies in the function `temperature` that takes a `Content` *signal* `normal`, a conversion function `f`, another `Content` signal `other` and returns a `Content` signal. The function `temperature` is responsible for handling the conversion in one direction which is why we later apply it twice in the definition of `main` to achieve the desired bidirectional behavior. In the body of `temperature` we define a helper variable `conversions` which is a signal of a conversion function. The `merge` combinator creates a signal c from two source signals a and b that emits the same value as a whenever a emits and the same value as b whenever b emits. In this case a is the constant signal function[46] returning `normal`'s current value (a `Content`) unchanged regardless of what input value it receives and b is the signal function that converts the value of other (a `Content`) into the unit of `normal`[47]. Let us suppose that `normal` is `celsius.signal` and `other` is `fahrenheit.signal`. Then a emits whenever the user changes the string in the Celsius input field[48] and accordingly `conversions` emits a function that, no matter its argument, simply returns the same value a has emitted (which is the current user-entered string in the Celsius field). The signal b, in contrast, emits whenever the user changes the string in the Fahrenheit input field and accordingly `conversions` emits a function that either ignores its argument and converts b's emitted `Content` into Celsius if said `Content` was numeric. Otherwise, its argument, which should be the current resp. last `Content` of the Celsius field, is returned unchanged. Only in this situation is the argument to `conversions` ever used. So, if `conversions` received as its argument the *last* `Content` of the Celsius field[49] then we could achieve the desired behavior (in one direction). Enter `foldp`[50]. The signal combinator

[46] `always` creates a constant function, i.e. a function that always returns the same value regardless of input. It is defined as `always a b = a`. The second argument is hence ignored.

[47] Remember the semantics of `convert`. If `other`'s value is numeric then it is converted as per `f` otherwise the value of b's argument is returned unchanged.

[48] Which is why `display` needs a handle for these fields in order to assign user interactions to the respective signal. As a side mark, it seems a bit unfunctional that `field` needs a handle but there is probably no way around that.

[49] *Not* solely the last string issued by the *user* into the Celsius Field. It is intricate to explain but this is the reason why `(<|) <~ conversions ~ normal` would not suffice. Suppose the user enters "0" into the Celsius field (which results in "32" in the Fahrenheit field). Then he changes the string in the Fahrenheit field to "104" (which results in "40" in the Celsius field). Now, he enters "104x" into the Fahrenheit field which is a non-numeric string. We want to have the Celsius field keep "40" in this case but without `foldp` we would see "0" in the Celsius field, the last entered string into the Celsius field *by the user*. Put another way, our field signals only emit in reaction to user interactions and not when we simply programmatically "paint over" a field with a new value and this is the reason we need to keep the "painted over" `Content`s with `foldp`.

[50] As a remark, this combinator is central to Elm (or FRP systems in general). What

`foldp` is similar to the `accumulate` combinator from ReactFX. We have a transition function g, an initial accumulator value x and an succession of values $y_1, \ldots, y_n$ from the signal y and return the signal value that results from $g(y_n, g(\ldots, g(y_1, x)\ldots))$. Whereas `merge`'s role was the combination of signals so is `foldp`'s role the past-dependent transformation of a signal. In our case, the transition function g is the apply function `(<|)`[51], the initial value x is the empty `Content` and the signal value y is the conversion function represented by `conversions`. So, for a succession $c_1, \ldots, c_n$ of conversion function values from the signal `conversions`, `foldp`'s returned signal value would be

$$\text{apply}(c_n, \text{apply}(\ldots, \text{apply}(c_1, \text{noContent})\ldots))$$

which is nothing else than

$$c_n(c_{n-1}(\ldots c_1(\text{noContent})\ldots)).$$

For the direction from Fahrenheit to Celsius `foldp`'s signal value is thus always the `Content` of the Celsius field either as converted by the corresponding Fahrenheit value or as entered into the Celsius field by the user.

The function `display` is much tamer than `temperature`. It takes a handle and a `Content` to show for each the Celsius and Fahrenheit temperature. It horizontally lays out the Celsius and Fahrenheit input fields together with the respective labels to make the resulting application look as depicted in figure 3.2.

Finally, all is brought together in the definition of the variable `main`. As before we have the `frame` function that takes a title and the element[52] to show inside a frame. The variable `fields` is a signal of the element we want to have inside the frame. It is simply the result of `display` applied to the right arguments. That is, our two field handles and the signals of `Content` for one and the other direction[53] [54].

As can be seen by the length and involvedness of the explanation our solution with Elm is very exotic. This worsens the Abstraction Level drastically because the typical programmer will need to expend a lot of learnability effort to become productive with Elm[55]. To really understand the `temperature`

can be done with encapsulated state in OOP can often be achieved with `foldp` in Elm.

[51] We could have equivalently written the following lambda expression: `(\f x -> f x)` which is equivalent to the following Scala expression `(f, x) => f(x)`.

[52] Which can of course be composed of other elements like in our situation.

[53] `f <~ a ~ b` is equivalent to `lift2 f a b` where `f` is a function of two arguments and `lift2` lifts a function of two arguments into the world of signals. That is, the type of `lift2` is `(a -> b -> c) -> Signal a -> Signal b -> Signal c`.

[54] In our implementation there are no feedback loops because, roughly speaking, our field signals only emit in reaction to user interactions and not when we programmatically "paint over" a field (i.e. just replace its `Content`).

[55] Elm's promise though is that this is a constant cost and so once it is spent all future tasks will not seem exotic anymore.

function one must be comfortable with higher-order programming and the combinators `merge` and `foldp`. These are central to Elm so they must be understood one way or another in order to become productive with Elm. But especially `foldp` has a high Abstraction Level. On the other hand, this combinator allows us to model many things which would need (encapsulated) state in OOP so it is not only hard to learn but also versatile. The Diffuseness is comparable to the other solutions. The separation between an `Input` and its `Content` has an arguably worse Closeness of Mapping than simply thinking of both as an encapsulated unit which also in the first place leads to the need to use `foldp`. In other examples it may be true that by this separation the view and model more naturally separate, too, and the view function(s) become pure and easier to write. But in this example, especially compared to our previous solutions, this does not seem to be the case. In our solution and in Elm in general Hidden Dependencies are not a problem as the pure functional programming style with its focus on declaring *all* function inputs as parameters and the separation of signals and non-signals makes dependencies very explicit[56]. The notation does not invite mistakes. Combined with Elm's static type system Error-Proneness is good. As found out in the previous case study Elm in general has a relatively high Viscosity due to the separation of signals from non-signals but for Temperature Converter we did not find additional aspects that would increase Viscosity.

If we compare the solution with the solution in pure ScalaFX then ScalaFX's solution is much easier to understand but also less general due to the use of the special case bidirectional binding construct. The solution with Scala.Rx is arguably as general as the solution in Elm yet seems to be much easier to understand, too. The reason is that it is more convenient to directly refer to the input fields in the binding expressions without a need to distinguish between signals and non-signals as well as without the need to distinguish between just "painting over" an input field versus directly setting its (encapsulated) text value. Even though the solution in ReactFX uses (stream) combinators, too, it gets along with only using the more basic combinators `map` and `filter` and otherwise has the same advantages over the solution in Elm as the solution in Scala.Rx and pure ScalaFX. To sum up, in this small case study Elm could not show (yet) that its alternative approach pays off. It might be the case that bigger examples exhibit more receptivity for the Elm approach.

The next case study is Flight Booker (3.5). The solution in Elm is as follows[57].

[56] Although, again, one has to be careful with the impure `input` function. Initially, the function `display` only took the arguments `c` and `f` and used `celsius.handle` and `fahrenheit.handle` directly in its body which can be considered a hidden dependency. Admittedly, the other solutions in essence have exactly this dependency.

[57] The solution presented here was presented to Elm's mailing list and deemed to be

```
data FlightType = OneWay | Return

flightType = input OneWay
startDate  = input { noContent | string <- dateToStr (fromTime 0) }
returnDate = input { noContent | string <- dateToStr (fromTime 0) }
book = input ()

display fHandle sHandle rHandle bHandle f sContent rContent =
  let s = sContent.string
      r = rContent.string
      dateField str = if isDateString str then field else redField
  in
      vbox
      [ dropDown fHandle
          [ ("one-way flight", OneWay)
          , ("return flight",  Return)
          ]
      , dateField s sHandle id "Start date" sContent
      , if f == Return
            then dateField r rHandle  id "Return date" rContent
            else label "disabled"
      , if f == OneWay && isDateString s ||
               isDateString s && isDateString r && strToDate s `before` strToDate r
            then button bHandle () "Book"
            else label "No Book!"
      ]

main =
  let content = display flightType.handle startDate.handle returnDate.handle book.handle
                        <~ flightType.signal
                         ~ startDate.signal
                         ~ returnDate.signal
  in  frame "Flight Booker" <~ content
```

The first line might stand out. With it we define an *Algebraic Data Type* (ADT) `FlightType` which has two possible variants `OneWay` and `Return`. In this case, we can simply think of it as an enum as found in Java[58]. As usual, we need to define the necessary signal `Inputs`. The input `flightType` represents the selected flight type from the combo box C which has the initial value `OneWay`. The inputs `startDate` and `returnDate` represent the contents of the corresponding text fields T_1 and T_2. We set their initial value to "1970.01.01"[59]. The input `book` represents the book button B's click event[60].

representative Elm code. As was the case for all the previous solutions, helper functions are omitted.

[58] We could have used strings instead of an ADT like in the other solutions for this case study. But using ADTs this way is more representative for Elm code and ADTs are needed again in the following (more complicated) solutions.

[59] This is the result of `dateToStr (fromTime 0)` where `fromTime` takes a UNIX time, i.e. a date and time combination encoded as a float. We will later go into why we do not use the current date as the initial value like in the other solutions.

[60] We are not actually interested in this event since we do not handle it in this case

The next function `display` already describes the view. It takes the respective handles, the selected `FlightType f` and the `Contents sContent` & `rContent` of the text fields T_1 & T_2, respectively. The variables `s` and `r` are simply shortcuts. The function `dateField` takes a string value `str` and creates an ordinary text field function if `str` is a correctly formatted date string and otherwise returns a text field function whose created text field's background is red. In the body we create the layout of the resulting application *and* at the same time implement the functional dependencies between the widgets as demanded by the case study. The layout combinator `vbox` is analogous to the already established `hbox` combinator with the difference that it lays out its list of elements vertically instead of horizontally. The first element is a combo box resp. drop down created by the function `dropDown` that takes a corresponding handle and a list of options. Each option is a tuple where the first component is the shown string and the second component the value the corresponding input signal will be set to when the user selects this option. The next element is the (date) text field for the start date[61]. Note that by using `dateField` instead of simply `field` we automatically implement the constraint that the text field will be red for a malformed string and otherwise white. The following element is the return date field. As there is no default way in Elm to have a disabled widget we simply show resp. draw a label with the string "disabled" when the selected flight type is not a return flight and otherwise show the respective `dateField` for the return date. Again, note that with this `if` expression and the usage of `dateField` we have implemented both constraints for the return date text field. The final element is the book button which is expressed similarly to the return date field with the difference that the boolean expression is slightly more involved[62]. As before, all is brought together in the definition of the `main` variable.

The aspect that makes this solution unique compared to our previous solutions (in Elm and also JavaFX/ScalaFX) is the fact that the implementation of the functional dependencies is inside the draw function `display`. This "immediate-mode inspired"[63] way of drawing the GUI makes the description of

study but the Elm function `button` which creates a button widget expects an input handle. This is why we created this input in the first place and simply initialize it with `()` which is the canonical "empty value".

[61] The function `field` has already been explained in our solution for Temperature Converter. The function `dateField` is identical in that respect.

[62] The details of this expression will not be covered because they should be already familiar from the previous solutions. One syntactical curiosity might be `` `before` ``. The back ticks are simply a way to use a binary function like a binary operator, i.e. it is equivalent to `before (strToDate s) (strToDate r)`.

[63] There are legitimate concerns with respect to animations in and performance of this approach. It is not exactly clear how this approach would handle a slightly altered task where e.g. the background of a text field should slowly fade into red. Looking at the `display` function it seems that the GUI is simply redrawn as a whole on every change. In reality, a clever diffing scheme is used to compute the minimal changes to apply to

functional dependencies in code very natural as they are expressed solely with basic and native language constructs. Roughly speaking, we get Scala.Rx's good way of describing functional dependencies for free. Therefore, the Abstraction Level is very good in this respect as is the Closeness of Mapping because the idea of simply (re)drawing the GUI like one would draw a picture is very easy to grasp. The Diffuseness is very good as well since there is no boilerplate at all required to implement the functional dependencies. The remaining dimensions are good, too, but there is nothing particular to say about them[64].

However, we need to mention now why we chose to use a fixed initialization date for the text fields. If we wanted to initialize e.g. `startDate` with the *current* date and not with an arbitrary, we would have to add a new parameter (the current date) to display which would constitute the initial value for `startDate` and we would need to use `foldp` to figure out if startDate has not been changed yet by the user. The changes would look something as follows.

```
startDate = input noContent
display ... sInitial sUntouched =
  ...
  dateField s sHandle id "Start date" (if sUntouched then sInitial else sContent)
  ...
main =
  let currentDate = dateToContent . Date.fromTime  <~ programStarted
    sUntouched = foldp (always False) True startDate.signal
    content = display ... <~ ... ~ currentDate ~ sUntouched
```

This seems very involved and the signal `programStarted` does not exist in Elm actually. Also, it is not clear if this solution would really work. In Elm's mailing list it was suggested to basically use JavaScript to circumvent this difficulty. The point is that pureness in the form of the signal/value separation breaks apart for this relatively harmless requirement which worsens the Abstraction Level as we need to circumvent the signal/value separation abstraction since it is too restricted. If we turn a blind eye to this issue we can still conclude that this solution in Elm for the Flight Booker is quite good.

The next case study we want to take a look at is Crud (3.7). We will

the view. This is the approach prominently taken and explained in greater detail by the framework http://facebook.github.io/react/. But we did not test these concerns in this case study. In any case, seeing as other popular frameworks employ this technique, like React or Om, it definitely has its merits.

[64] One minor concern one could raise is that the layout definition is intertwined with the definition of the functional dependencies. However, this concern really is minor because one could easily put, e.g., the if expressions into the `let` head and give them a shorter name to be used in the `let` body and as such achieve a separation of layout and functional dependencies if one wanted to.

present the source code of the solution in Elm in small pieces[65]. The first piece of code we show is the "Model" which is a representation of the application domain.

```
type Entry = String
type CrudState = Array Entry
initialState = Array.fromList ["Emil, Hans", "Mustermann, Max", "Tisch, Roman"]
data Action = Create Entry | Update Int Entry | Delete Int | None
```

The `type X = Y` syntax introduces `X` as an alias for `Y`. An entry in our CRUD application is therefore simply a string. The relevant state of our application is then an array of entries with the initial state being an array of the already familiar exemplary names. The ADT `Action` describes the operations the user can perform which are creating a new entry, updating an entry at a specified index, deleting an entry at a specified index and simply doing nothing. The next piece addresses the updating of the state based on user's actions.

```
deleteArrayElement index array = -- ...
update : Action -> CrudState -> CrudState
update action state =
  case action of
    None       -> state
    Create e   -> Array.push e state
    Update i e -> Array.set i e state
    Delete i   -> deleteEntry i state
```

`deleteArrayElement` is a helper function that given an index i and an array $(x_0, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_m)$ creates a new array $(x_0, \ldots, x_{i-1}, x_{i+1}, \ldots, x_m)$. The function `update` takes an `Action` and a (or rather the current) state of the application and creates a new updated state. For each case of a user's action the exact operation differs. In the `None` case the new state is unchanged. In the `Create` case the entry `e` is added to the end of the list of entries (the application state). In the `Update` case the entry at index `i` is replaced with `e` and in the `Delete` case the entry at `i` is simply deleted. The next piece describes the GUI input elements.

```
prefixInput  = input Field.noContent
nameInput    = input Field.noContent
surnameInput = input Field.noContent
actionInput  = input None
selectedInput : Input (Maybe Int)
selectedInput = input Nothing
```

The first four inputs should be clear from the case study's description and the knowledge we already gained from the previous solutions in Elm.

[65] The solution presented here was accepted as representative Elm code in Elm's mailing list and is modelled after a somewhat similar Elm program: https://github.com/evancz/TodoFRP. Note that, normally, we would be using a module structure as outlined in appendix D but refrained from doing so for reasons of simplicity.

`selectedInput` represents the index of the selected entry and is of type `Maybe Int`. That means, it can have the value `Nothing` which stands for no selection or `Just i` which means that the entry at index `i` is selected. Now, the main part, the view, of the code is presented which describes how to display the model and the inputs on screen.

```
displayNameInputs name surname = -- uninteresting

displayEntry : Entry -> Int -> Maybe Int -> Element
displayEntry entry index selected =
  let box x     = container 200 22 midLeft x
      blueBox x = color blue (box x)
      txt       = plainText entry
  in
    if Just index == selected
      then blueBox txt |> clickable selectedInput.handle Nothing
      else box txt     |> clickable selectedInput.handle (Just index)
```

`displayNameInputs` simply draws the name/surname text fields together with their corresponding labels as shown in figure 3.5 – nothing unusual. `displayEntry` takes an `Entry`, an `index` of type `Int`, another index `selected` of type `Maybe Int` (suggesting that nothing can be selected) and creates a textual element representing the given entry resp. name. `container` creates a rectangular shape of a certain width and height and given an alignment puts its last argument (here `x`, an `Element`) inside this shape. If the index is equal to the selected index then `displayEntry` will create an entry with a blue background indicating the selection to the user. The function `clickable` takes a handle to an input and updates the respective signal to its second argument whenever the user clicks its third argument[66]. So, whenever the user clicks on an entry in the list of entries it will become the selected entry and the signal value of `selectedInput` will be updated to its index. And when a user clicks on an already selected entry, the selection will be removed.

```
displayEntries entries selected =
  container 200 300 topLeft
    <| flow down
    <| map (\(i, x) -> displayEntry x i selected) entries
```

`displayEntries` is responsible for not only drawing one entry but the complete list of entries. It takes an indexed list of entries and the selected index. It creates a container that contains the list of entries. `flow down` has a list of elements flow down. The list of elements is described by the map expression which transforms every element in the list `entries` as described in its lambda expression, i.e. `displayEntry` is applied to each element.

```
display : Content -> Content -> Content -> Maybe Int -> CrudState -> Element
display prefix name surname selected state =
```

[66] The operator `|>` is reversed function application, i.e. `a |> b` is equivalent to `b a`.

```
  let fullname    = surname.string ++ ", " ++ name.string
      filtered    = filter (\(i,e) -> startsWith prefix.string e) (toIndexedList state)
      filtIndices = Array.fromList <| fst (unzip filtered)
      filt2orig x = Array.get x filtIndices
      origIndex   = maybe Nothing filt2orig selected
  in
    vbox
    [ hbox [ label "Filter prefix: ", field prefixInput.handle id "" prefix ]
    , hbox [ displayEntries filtered selected, displayNameInputs name surname ]
    , hbox [ button actionInput.handle (Create fullname) "Create"
             , case origIndex of
                  Nothing -> label "No Update"
                  Just i  -> button actionInput.handle (Update i fullname) "Update"
             , case origIndex of
                  Nothing -> label "No Delete"
                  Just i  -> button actionInput.handle (Delete i) "Delete"
             ]
    ]
```

The heart of this solution lies in the function `display` that takes a prefix, name and surname all of type `Content`, the selected Index of type `Maybe Int` and the list of entries of type `CrudState` and creates the view of the GUI application. In the `let` head we define several helper variables. `fullname` should be clear. `filtered` is an indexed list[67] of the entries filtered to contain only those whose second component starts with the given prefix. `filtIndices` is the same list as `filtered` but its elements only consist of the first tuple component. `filt2orig` is a function that maps the index of an entry in a possibly filtered list of entries back to its index as it were in the unfiltered state of the list. If `x` is out of bounds it maps to `Nothing` otherwise to `Just` y where y is the original index. This function is needed as the user actions work on the whole list of entries and not just on a filtered view/list of the entries. `origIndex` represents the selected index but as it were in the unfiltered list. If `selected` is `Nothing` (i.e. no entry is selected) then `origIndex` is `Nothing` as well and otherwise it has the same value as `filt2orig` applied to the selected index. In the body of the `let` expression we define the layout. Everything should be clear as there are no new concepts compared to the previous solutions. We use the same "trick" as in the Flight Booker solution where we simply draw a label instead of the button if the button is disabled[68]. As always, we bring everything together in the variable `main`.

```
currentState : Signal CrudState
currentState = foldp update initialState actionInput.signal

main = let content = display <~ prefixInput.signal  ~ nameInput.signal
```

[67] That is, a list of tuples where the first component is the index of the second component.

[68] Note that we deliberately used handles in a global way as listing them as function inputs would have been too inconvenient.

```
                          ~ surnameInput.signal ~ selectedInput.signal
                          ~ currentState
    in frame "CRUD" <~ content
```

`main` is as usual but the interesting bit is the signal `currentState`. Using `foldp` we describe that the accumulated value is initialized with the initial state and is updated whenever the user performs an operation.

We took some liberties in this solution as Elm is still relatively young. For example, the external database aspect from the other solutions is not present and we did not use a border layout. This should be kept in mind for the following analysis.

The approach of this solution, especially the high-level structure of the code, differs from the other solutions. Whereas the state was encapsulated in the observable list objects `db`, `dbView` and the list view object `entries`[69] in the JavaFX (ScalaFX) solution, here the state is explicit, global, centralized and not encapsulated. Whereas we used JavaFX properties and binding expressions to formulate functional dependencies between widgets, here – due to the immediate-mode style of drawing the GUI– there is no need for these concepts[70]. Whereas we directly manipulated the state through callbacks on the buttons, here we use Elm's `Input` style where we set the global user action signal as a result of a button click and inside a central step or update function the explicit state is transformed[71].

As we do not need the concept of properties to express functional dependencies the Abstraction Level in that respect is better. In JavaFX the framework provided (conceptually) the `filt2orig` function but in our Elm solution it is quite easy to define this function yourself. Instead of callbacks on the buttons we use Elm's `Input` approach. However, it is not clear which solution has a better Abstraction Level in this regard. The same is true for the contrast between explicit and centralized state versus encapsulated state. Making the state explicit and centralized as well as the user actions, however, improves the dimension Hidden Dependencies. On the other hand, Diffuseness and Viscosity are increased because state has to be pipelined through function calls[72]. As the state and other aspects are explicit and centralized there is less possibility for it to become inconsistent due to accidental changes to it from more remote code and therefore Error-Proneness

[69] `entries` encapsulated the selection state.

[70] As we already established in the analysis of the previous solution for `Flight Booker`.

[71] As a side mark, the approach of making an element clickable (with the function `clickable`) in Elm is also somewhat interesting. In an OOP framework an object typically has a list of event listeners and we can register an event listener/callback for clicks. The function `clickable` that takes an element and creates a new clickable element is more reminiscent of the object-oriented Decorator pattern which in OOP GUI frameworks (like Swing) sometimes is used for "decorations" like borders.

[72] In other pure functional languages like Haskell, monads are a way to circumvent this problem.

is improved. Although the approach taken in this solution is considerably different from the solution in JavaFX the Closeness of Mapping is arguably neither better nor worse – just different. On the whole, the Diffuseness is similar to the other solutions. All in all, this is a good solution for the CRUD case study[73].

The time unfortunately ran out for the other case studies. It seems that hypothetical Elm solutions for TIMER and CIRCLE DRAWER would be similar to the solutions in ReactFX. We hope this section still provided a more precise idea of Elm's (and similar more "radical" FRP GUI toolkits) approach and its pros and cons.

We saw that Elm requires a considerable learning effort for the typical programmer and that some easy things with JavaFX[74] required a lot of work in Elm. On the other hand, we saw that Elm's "immediate-mode inspired" way of describing the view part of the GUI has real benefits for expressing functional dependencies. Furthermore, Elm's approach of having centralized and explicit state instead of encapsulated state and the resulting code structure thereof can be advantageous.

Elm is still quite young and, e.g., it is not yet clear for the author of Elm if the current solution of handling input elements like buttons is ideal. So there is the possibility of changes resp. improvements over the current approach and it will be interesting to see how GUI programming with Elm evolves in the future.

[73] Of course, since the external database aspect has not been considered as opposed to the solution in JavaFX it could very well be the case that the performance of this solution will not be sufficient for large datasets. In principle, this state centralizing approach could also be used relatively pain-free in an OOP language but encapsulated state is generally preferred.

[74] Or ScalaFX, Scala.Rx and ReactFX.

4.4 Verdict

In this chapter we have examined FRP approaches to our case studies in the form of three different libraries or, in the case of Elm, even languages. In the section on Scala.Rx we have seen that its syntactical approach to describing functional dependencies with automatic dependency derivation and tracking is more convenient than the current approach employed by JavaFX (ScalaFX). ReactFX as a Cousin of FRP specifically aimed at GUI programming showed that it can provide benefits when the problem involves something that is naturally represented as events (e.g. mouse clicks, timers) and substantial benefits when the problem involves asynchronous requirements. Elm shares advantages of ReactFX but also provides some unique ideas that show promise as well. Having pure functions that draw the GUI in an immediate-mode inspired way makes implementing functional dependencies very easy. Centralizing the program state and making it explicit with the associated consequences to the program structure showed promise with respect to preventing unintentional mistakes.

In the case of ReactFX, and considerably moreso in the case of Elm, the Abstraction Level in general is more demanding and the dimensions Diffuseness and Viscosity are slightly higher compared to the solutions from chapter 3. On the other hand, Hidden Dependencies and Error-Proneness are reduced. For ReactFX the reason for this trade-off is that describing interactions with event streams makes it possible to use mainly side-effect free event stream compositions to model behaviour and only employ side-effecting operations at the edge of the system. Although often natural there is still the need to express solutions in terms of the event stream combinator language which can be considered a small world in itself. However, the mental effort to understand this way of structuring code is relatively low, particlularly compared to Elm. Elm shares the same benefit with respect to event stream composition. Yet, being a pure functional language, Elm goes further and employs more constraints – most notably in the form of separating signals from other values. Arguably, Error-Proneness is further reduced but for the price of higher conceptual costs.

The most versatile, practical and immediately useful form of FRP for improving certain GUI programming challenges, particularly with respect to providing a better alternative to callback heavy code, seems to be a Cousin of FRP like ReactFX. ReactFX, being primarily written for Java, shows that its FRP approach can be introduced even in an imperative OOP language on top of an existing object-oriented GUI toolkit like JavaFX. To reap the benefits of Scala.Rx's approach, the host language must provide some more syntactical expressiveness than Java. With both Scala.Rx and, in particular, ReactFX FRP solutions can be gradually incorporated into certain parts of the program where they make sense. Although Elm also provides interoperability with its host systems JavaScript, HTML and CSS, it seems that Elm's interoperability

would be in the form that a whole component is written in Elm which can then be embedded into a larger application, but a more fine-grained and still idiomatic interoperability as with e.g. JavaFX and ReactFX seems harder to achieve due to Elm's additional constraints on program structure.

The three presented projects are quite young and still more experience needs to be gathered to get a clearer picture of FRP's practical applicability to GUI programming in its various manifestations. A much tighter integration into existing and widespread GUI toolkits[75] should be focused on to lower the barriers of playing with FRP in this context. We showed how an exemplary tighter integration into ScalaFX can be achieved using Scala's Implicits but there are still practical problems with this approach that will hopefully be ironed out in a future Scala release.

If we compare the general approach of FRP to OOP we can make and have made the following observations that are shared by others. "OOP makes state explicit but encapsulates it, wheras state in FRP is hidden from the programmer by the temporal abstractions of the language" [22]. Especially in Elm's solution for the CRUD case study it was "noticed that signal creation feels very much like a restricted form of OOP, where all objects have to conform to a strict interface. The restrictions make it possible to avoid certain pitfalls of traditional OOP, e.g. being able to modify the state of an object from several different sources without having to somehow coordinate those changes" [56]. A valid question is if this restricted expressivity and rethinking effort combined with potential reduction in error proneness will be accepted by typical programmers and worth it for typical programs. Particularly Elm with its more radical approach and insistence on separation between conventional values and signals might be too high a barrier for most programmers or programs. Again, an interesting opinion is given by [56]: "There's a trade-off between putting too many constraints over the way programs can be structured versus letting the programmers shoot themselves in the foot (often in subtle ways that might take a long time to realise), and the stricter end of the spectrum is of course the less popular one. In the end, I'm not expecting FRP to get big – not in its more rigorous forms at least –, but I still recommend studying it for the valuable lessons that can make our software better, no matter what approach we use."

We think it would be also worthwhile to focus or find more negative GUI programming examples to uncover or dismiss some potential limitations of (Siblings of) FRP for example with respect to a static dependency graph. Perhaps, the list of case studies from this thesis could be extended with one that pushed FRP solutions to their limits. An exemplary question from [29] is: "How would one write a code editor with FRP?". Another open problem is (better) support for multidirectionality [5], i.e. cyclic data flow dependencies,

[75] Which is already the case for ReactFX as it is primarily developed for JavaFX. But ideally there should be much more tightly integrated FRP libraries into GUI toolkits.

which would have probably helped for the Temperature Converter case study in Elm. Yet another open problem is FRP's applicability to distributed programming [5] but this problem is less relevant for GUI programming. Note that most of these problems do not seem to exist in Cousins of FRP. There is also research of Reactive Programming applied to OOP languages [28, 60] although the practical availability seems to be behind current FRP projects. Still, it would be interesting to see how a reactive OOP language resp. library would fare for our case studies in comparison to the FRP solutions from this thesis.

Chapter 5

Conclusion

In this thesis we have compared and analyzed both OOP and FP solutions for a set of GUI case studies. In the introduction we wondered what role the paradigm played in contrast to the toolkit and which were the most beneficial features of either paradigm to increase the productivity of GUI programming. Thus, in chapter 3 we compared the OOP approach with the classical FP approach for our case studies. In chapter 4, on the other hand, we analyzed if the more modern FRP approach in its various forms might provide unique improvements both over the OOP as well as the classical FP approach.

We concluded in chapter 3 that the role of the toolkit dominated. Not only in the form of widgets and special-case constructs but also in the form of more general, paradigm-independent features like binding expressions and observable collections. Mostly paradigm-independent language features that facilitated GUI programming were succinct anonymous functions for callbacks and the ability to describe a GUI's layout with hierarchical expressions. Nevertheless, there were situations where the paradigm did make a difference. For example, FP's pervasive immutability simplified the implementation of undo/redo operations.

There were several additional insights gained in chapter 4. Using ReactFX's event streams to handle time- resp. behavior-related code as an alternative to callbacks is more explicit, less error-prone and well-suited for asynchronous requirements. Without a doubt, the results for our dimensions of this "Cousin of FRP approach" strongly suggest that it can increase GUI programming productivity, especially when it comes to the challenge of "juggling competing interactions with the GUI from direct user input but also from sensors and remote internet services" (see section 1.2).

A succinct way to describe functional dependencies between widgets is important as such dependencies are very common in GUI programming and reducing their overhead thus decreases verboseness of the resulting code. One way as shown by Scala.Rx is to essentially make binding expressions look as native and succinct as possible. Another way as shown by Elm is to simply

employ an immediate-mode inspired approach to GUI drawing and hence be able to describe functional dependencies "inline" for free.

Having state centralized – a "central store" – as in Elm together with immutability makes it even easier to implement undo/redo operations, potentially reduces error-proneness due to the ceased risk of different components or objects becoming inconsistent by uncoordinated state changes, and has less of the typical MVC-related problems as outlined in section 1.2. A central store actually goes quite directly against OOP's practice of encapsulating resp. distributing state into objects which have specially defined interfaces to get or alter part of the overall program state. Even with OOP's pitfalls, history shows that its approach scales to millions lines of code. Although the central store approach does seem convincing, too, it remains to be seen how it will handle bigger code bases. It might turn out that encapsulation of state really is not that beneficial[1] and, instead, "state-less" information hiding as outlined in appendix D is more responsible for OOP's success at scale[2]. At any rate, the usage of data bases as a central store in many OOP applications is indicative that Elm's approach might be a good one. On the other hand, this approach of structuring GUI programs is still quite young[3] and as such its very own potential pitfalls are yet to be discovered.

It also remains to be seen if the additional constraints of a purely functional language like Elm, i.e. with a separation of signals from other values, will gain widespread acceptance from programmers for its advantages or if its liabilities will prevent it. As a matter of fact, many of the beneficial features of an FP approach to GUI programming found in this thesis can be reused in an OOP language like Java[4]. For example, a library for persistent data structures could be employed to gain easier undo/redo functionality. ReactFX is an example of a Cousin of FRP working well together with Java and JavaFX. And even the state centralizing approach could be imitated with an immutable central store object. Granted, in a typical OOP language one must have more discipline when using these features so as not to, e.g., encapsulate part of the program in different objects instead of the central store one. A pure FP setting in many ways forces the programmer to maintain discipline for the price of less freedom in program structure. At the end of the day, the best approach for a specific project depends on many factors and a cost-benefit analysis should always be considered.

[1] Indeed, maybe even detrimental. Compared to unrestricted mutable global state, state encapsulation as in OOP definitely is an improvement. However, the problems of global mutable state might really lie in its mutability and not in its globality. So, state that is global and immutable might be a configuration that is even better (or not) than state that is encapsulated but potentially mutable.

[2] And of course many other characteristics of OOP as outlined in section 1.1.

[3] Other recent projects employing a central store approach for GUI programming are Om and Facebook's Flux.

[4] Although, it would be less idiomatic and convenient to use these FP approaches in Java than, for example, in Scala.

Hopefully, this thesis painted a clearer picture between the different approaches of modern OOP and modern FP variants to GUI programming and the role of the paradigm in contrast to the toolkit's. Perhaps even new insights were gained which previously only have been known more or less exclusively to certain pioneering groups. A very condensed assessment of the most important concepts examined in this thesis is given in table 5.1. In any case, the current activity of novel GUI programming ideas, particularly in the web development field, is exciting to follow and it will be very interesting to see which ideas will come out on top in the future and if one of them will be an approach discussed in this thesis.

Dimension	Callbacks	Streams	Bindings	Imm.Mode	Hier.Expr.
Abst.Lev.	+	+	++	++	++
Clos.o.M.	o	+	++	++	++
Hidd.Dep.	−−	+	++	++	++
Err.Pron.	−	++	+	++	++
Diffuseness	+	o	+	++	++
Viscosity	+	+	+	++	++
Recommend	for deliberate side-effects	yes	yes	yes, but depends on framework	yes

Dimension	Persistence	Obs.Collect.	State Enc.	Centr.Store	Pureness
Abst.Lev.	+	++	o	+	−
Clos.o.M.	o	++	+	o	o
Hidd.Dep.	++	++	o	++	++
Err.Pron.	++	++	o	++	++
Diffuseness	+	++	+	+	o
Viscosity	+	++	++	−	−
Recommend	yes	yes	yes	maybe, worth a try	if error prevention of utmost importance

Figure 5.1: Takeaway for usefulness of the concepts examined in this thesis.

Chapter 6

Further Work

The Cognitive Notations of Dimensions framework has turned out to be very helpful in analyzing the various paradigms in this thesis. Not only did the dimensions provide a stable platform for discussion but each dimension also allowed for many interesting points to be made about the different solutions. More value should be placed on identifying usability aspects of programming languages as programming becomes more widespread. The Cognitive Notations of Dimensions framework can be used as a tool to at least cover the notational usability aspects. After first insights have been gained in an analytical fashion[1], it would make sense to follow up with empirical studies to give results more meaningfulness. In fact, an empirical study comparing program comprehension between classical OOP and reactive solutions has recently been conducted with the conclusion that Reactive Programming really does improve comprehension [59]. Ideally, more such studies should be conducted to have better arguments for and against certain approaches or paradigms.

The systematic analysis of solutions based on a fixed set of manageable case studies turned out to be useful, too. Such an approach could be used as a basis for further analyses. The following additional aspects could be taken into consideration for an extended resp. modified set of case studies:

- An asynchronous requirement[2]
- Animations and animated transitions[3]
- Drag and drop[4]
- A requirement that is difficult to implement with a static dependency

[1] As is the case for this thesis.

[2] As presented at the end of section 4.2 but smaller in scope.

[3] Particularly on mobile platforms, animations become increasingly important. A possibility would be to modify Flight Booker to have the additional requirement that the fields must slowly fade to red.

[4] Circle Drawer could be extended with the requirement that the circles be draggable. Drag and drop is a motivating example from [28].

graph resp. with current Siblings of FRP[5]

- A requirement that tests for the absence of glitches[6]

Of course, the difficulty lies in incorporating these additional aspects in as self-contained as possible case studies resp. modifying the existing ones without bloating them up too much.

Another strategy is to instead consider much bigger case studies in order to find out if an approach scales. Naturally, such case studies would not be examined comprehensively in detail. Nevertheless, statistical metrics could be applied to judge the quality of a solution. This is certainly already done in practice [34]. Combined with our strategy and empirical studies it is doubtlessly possible to paint a much clearer picture about the utility of specific programming approaches.

Currently, there is much activity in the web development world with respect to browser-based GUI frameworks. It would be interesting to create some kind of taxonomy of these frameworks and systematically compare their viability. In fact, there is a de facto standard notational benchmark for browser-based GUI frameworks, namely TodoMVC [57]. TodoMVC is quite in the spirit of our case studies with the difference that it is a single and bigger one instead of a set of case studies and has more of a focus on web-related challenges. TodoMVC, perhaps together with all or parts of the case studies from this thesis, could be used as a basis for a comprehensive overview of the browser-based GUI framework landscape.

In the following we want to give an outlook on research directions or activities with respect to Reactive Programming in the context of GUIs since the Reactive Programming paradigm seems to be most promising to further improve the development of GUIs. In the Haskell community there is a recent effort to create a variation of FRP, namely "state-based FRP", whose aim is to be better suited for GUI programming than former FRP systems [58]. In this thesis we have seen ReactFX as an example of an FRP library that works very well together with an object-oriented GUI toolkit. Indeed, FRP and OOP can go together as several papers show [28, 32, 34]. Some ideas from Scala.React [28], especially with regard to the imperative data-flow language, are very interesting but have not yet been taken up again for some reason. If we consider Reactive Programming without the F then we can find other interesting projects[7]. For example, in "Programming with Managed Time" [29] the authors present the system Glitch which makes it possible to reap the benefits of reactivity without giving up the familiar imperative paradigm. Many other interesting open questions and research directions can be found in [60]. A primary goal is to integrate Reactive Programming well

[5] For example, a run-time dependent number of reactive dependencies should be created in some way.

[6] See appendix E for a short overview on glitches.

[7] Albeit at relatively early stages.

into OOP. Clearly, the Reactive Programming space is encouraging and there is yet much to be discovered.

References

Literature

[1] Jonathan Aldrich. "The Power of Interoperability: Why Objects Are Inevitable". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! '13. Indianapolis, Indiana, USA: ACM, 2013, pp. 101–116. URL: http://doi.acm.org/10.1145/2509578.2514738 (cit. on pp. 2, 4).

[2] C. Appert and M. Beaudouin-Lafon. "SwingStates: adding state machines to Java and the Swing toolkit". In: *Software: Practice and Experience* 38.11 (2008), pp. 1149–1182. URL: http://dx.doi.org/10.1002/spe.867 (cit. on p. 9).

[3] Deborah J. Armstrong. "The Quarks of Object-oriented Development". In: *Commun. ACM* 49.2 (Feb. 2006), pp. 123–128. URL: http://doi.acm.org/10.1145/1113034.1113040 (cit. on p. 4).

[4] John Backus. "Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 613–641. URL: http://doi.acm.org/10.1145/359576.359579 (cit. on p. 4).

[5] Engineer Bainomugisha et al. "A Survey on Reactive Programming". In: *ACM Comput. Surv.* 45.4 (Aug. 2013), 52:1–52:34. URL: http://doi.acm.org/10.1145/2501654.2501666 (cit. on pp. 59, 68, 98, 99, 118).

[6] Judith Bishop and Nigel Horspool. "Developing Principles of GUI Programming Using Views". In: *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '04. Norfolk, Virginia, USA: ACM, 2004, pp. 373–377. URL: http://doi.acm.org/10.1145/971300.971429 (cit. on p. 8).

[7] Magnus Carlsson and Thomas Hallgren. "FUDGETS: A Graphical User Interface in a Lazy Functional Language". In: *Proceedings of the Conference on Functional Programming Languages and Computer*

Architecture. FPCA '93. Copenhagen, Denmark: ACM, 1993, pp. 321–330. URL: http://doi.acm.org/10.1145/165180.165228 (cit. on pp. 2, 9).

[8] Koen Claessen, Ton Vullinghs, and Erik Meijer. "Structuring Graphical Paradigms in TkGofer". In: *In Proc. of the International Conference on Functional Programming (ICFP'97*. ACM Press, 1997, pp. 251–262 (cit. on p. 9).

[9] Steven Clark and Curtis Becker. "Using the Cognitive Dimensions Framework to evaluate the usability of a class library". In: *Proc. Joint Conf. EASE & PPIG*. Ed. by M. Petre and B. Budgen. Apr. 2003, pp. 359–366 (cit. on p. 8).

[10] Gregory H. Cooper and Shriram Krishnamurthi. "Embedding Dynamic Dataflow in a Call-by-value Language". In: *Proceedings of the 15th European Conference on Programming Languages and Systems*. ESOP'06. Vienna, Austria: Springer-Verlag, 2006, pp. 294–308. URL: http://dx.doi.org/10.1007/11693024_20 (cit. on p. 59).

[11] Antony Courtney. "Functionally Modeled User Interfaces". In: *Interactive Systems. Design, Specification, and Verification*. Ed. by JoaquimA. Jorge, Nuno Jardim Nunes, and João Falcão e Cunha. Vol. 2844. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 107–123. URL: http://dx.doi.org/10.1007/978-3-540-39929-2_8 (cit. on p. 10).

[12] Antony Courtney and Conal Elliott. "Genuinely Functional User Interfaces". In: *Proceedings of the 2001 Haskell Workshop*. Sept. 2001 (cit. on pp. 2, 9).

[13] Evan Czaplicki. *Elm: Concurrent FRP for Functional GUIs*. 2012 (cit. on pp. 9, 80).

[14] Evan Czaplicki and Stephen Chong. "Asynchronous Functional Reactive Programming for GUIs". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: ACM, 2013, pp. 411–422. URL: http://doi.acm.org/10.1145/2491956.2462161 (cit. on pp. 2, 9).

[15] Ole-Johan Dahl. "Software Pioneers". In: ed. by Manfred Broy and Ernst Denert. New York, NY, USA: Springer-Verlag New York, Inc., 2002. Chap. The Roots of Object Orientation: The Simula Language, pp. 78–90. URL: http://dl.acm.org/citation.cfm?id=944331.944338 (cit. on p. 1).

[16] Jonathan Edwards. "Coherent Reaction". In: *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. OOPSLA '09. Orlando, Florida, USA: ACM, 2009, pp. 925–932. URL: http://doi.acm.org/10.1145/1639950.1640058 (cit. on p. 59).

[17] Conal Elliott and Paul Hudak. “Functional Reactive Animation”. In: *International Conference on Functional Programming*. 1997. URL: http://conal.net/papers/icfp97/ (cit. on pp. 58, 59).

[18] Erick Gallesio and Manuel Serrano. “Programming Graphical User Interfaces with Scheme”. In: *J. Funct. Program.* 13.5 (Sept. 2003), pp. 839–866. URL: http://dx.doi.org/10.1017/S0956796802004537 (cit. on p. 10).

[19] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995 (cit. on pp. 9, 45).

[20] Emden R Gansner and John H Reppy. “A multi-threaded higher-order user interface toolkit”. In: (). URL: http://alleystoughton.us/eXene/1993-trends.pdf (visited on 07/20/2014) (cit. on pp. 2, 9).

[21] Thomas R. G. Green. “Instructions and Descriptions: Some Cognitive Aspects of Programming and Similar Activities”. In: *Proceedings of the Working Conference on Advanced Visual Interfaces*. AVI ’00. Palermo, Italy: ACM, 2000, pp. 21–28. URL: http://doi.acm.org/10.1145/345513.345233 (cit. on p. 8).

[22] Daniel Ignatoff, Gregory H. Cooper, and Shriram Krishnamurthi. “Crossing State Lines: Adapting Object-oriented Frameworks to Functional Reactive Languages”. In: *Proceedings of the 8th International Conference on Functional and Logic Programming*. FLOPS’06. Fuji-Susono, Japan: Springer-Verlag, 2006, pp. 259–276. URL: http://dx.doi.org/10.1007/11737414_18 (cit. on pp. 10, 30, 98).

[23] Alexandros Karagkasidis. “Developing GUI Applications: Architectural Patterns Revisited.” In: *EuroPLoP*. Ed. by Till Schümmer. Vol. 610. CEUR Workshop Proceedings. CEUR-WS.org, 2008. URL: http://dblp.uni-trier.de/db/conf/europlop/europlop2008.html#Karagkasidis08 (cit. on pp. 5, 6, 9).

[24] Alan Kay. “History of Programming languages—II”. In: ed. by Thomas J. Bergin Jr. and Richard G. Gibson Jr. New York, NY, USA: ACM, 1996. Chap. The Early History of Smalltalk, pp. 511–598. URL: http://doi.acm.org/10.1145/234286.1057828 (cit. on p. 1).

[25] Alan Kay. “User Interface: A Personal View 1989”. In: (1989) (cit. on p. 1).

[26] Maria Kutar, Carol Britton, and Jonathan Wilson. *Cognitive Dimensions - An Experience Report* (cit. on p. 8).

[27] Daan Leijen. “wxHaskell: A Portable and Concise GUI Library for Haskell”. In: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*. Haskell ’04. Snowbird, Utah, USA: ACM, 2004, pp. 57–68. URL: http://doi.acm.org/10.1145/1017472.1017483 (cit. on pp. 2, 9).

[28] Ingo Maier and Martin Odersky. *Deprecating the Observer Pattern with Scala.React*. Tech. rep. 2012 (cit. on pp. 6, 9, 59, 60, 99, 103, 104).

[29] Sean McDirmid and Jonathan Edwards. "Programming with Managed Time". In: (). URL: http://research.microsoft.com/pubs/211297/managedtime.pdf (visited on 07/20/2014) (cit. on pp. 98, 104).

[30] Brad A. Myers. *Why are Human-Computer Interfaces Difficult to Design and Implement?* Tech. rep. 1993 (cit. on p. 5).

[31] Rob Noble and Colin Runciman. *Functional Languages and Graphical User Interfaces – a review and a case study*. 1994 (cit. on p. 9).

[32] John Peterson, Ken Roe, and Alan Cleary. "Practical Functional Reactive Programming". In: (). URL: http://www.cs.jhu.edu/~roe/padl2014.pdf (visited on 07/20/2014) (cit. on p. 104).

[33] Caitlin Sadowski and Sri Kurniawan. "Heuristic Evaluation of Programming Language Features: Two Parallel Programming Case Studies". In: *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*. PLATEAU '11. Portland, Oregon, USA: ACM, 2011, pp. 9–14. URL: http://doi.acm.org/10.1145/2089155.2089160 (cit. on p. 8).

[34] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. "REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications". In: *Modularity '14 - 13th International Conference on Modularity - 2014*. Lugano, Switzerland: ACM Press, Apr. 2014 (cit. on pp. 6, 9, 59, 104).

[35] Joel Brandt Spehen Oney Brad Myers. "InterState: A Language and Environment for Expressing Interface Behavior". In: (). URL: http://www.joelbrandt.org/publications/oney_uist2014_interstate.pdf (visited on 07/31/2014) (cit. on p. 9).

[36] Larry Tesler. "Object-oriented User Interfaces and Object-oriented Languages (Keynote Address)". In: *Proceedings of the 1983 ACM SIGSMALL Symposium on Personal and Small Computers*. SIGSMALL '83. San Diego, California, USA: ACM, 1983, pp. 3–5. URL: http://doi.acm.org/10.1145/800219.806644 (cit. on p. 1).

Online sources

[37] URL: https://github.com/lihaoyi/scala.rx (visited on 04/12/2014) (cit. on pp. 3, 61).

[38] URL: https://github.com/TomasMikula/ReactFX (visited on 05/12/2014) (cit. on pp. 3, 59, 68).

[39] URL: http://elm-lang.org/ (visited on 04/12/2014) (cit. on pp. 3, 9).

[40] URL: http://wcook.blogspot.de/2012/07/proposal-for-simplified-modern.html (cit. on p. 4).

[41] URL: http://en.wikipedia.org/wiki/Object-oriented_programming (visited on 03/04/2014) (cit. on p. 4).

[42] URL: http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html (visited on 03/04/2014) (cit. on p. 6).

[43] URL: http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/ (visited on 04/12/2014) (cit. on p. 8).

[44] URL: http://martinfowler.com/eaaDev/uiArchs.html (visited on 04/12/2014) (cit. on p. 9).

[45] URL: http://homepage.lnu.se/staff/daweaa/papers/2014WICSA.pdf (visited on 04/12/2014) (cit. on p. 9).

[46] URL: http://www.informatik.uni-bremen.de/~cxl/sml_tk/doc/manual.html (visited on 04/12/2014) (cit. on p. 9).

[47] URL: http://wcook.blogspot.de/2012/07/proposal-for-simplified-modern.html (cit. on p. 9).

[48] URL: https://github.com/kentuckyfriedtakahe/sodium (cit. on p. 9).

[49] URL: http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html (visited on 04/12/2014) (cit. on pp. 10, 15).

[50] URL: http://en.wikipedia.org/wiki/Reactive_programming (visited on 05/12/2014) (cit. on p. 58).

[51] URL: http://en.wikipedia.org/wiki/Functional_reactive_programming (visited on 05/12/2014) (cit. on p. 58).

[52] URL: https://rx.codeplex.com/ (visited on 05/12/2014) (cit. on p. 59).

[53] URL: https://github.com/Netflix/RxJava (visited on 05/12/2014) (cit. on p. 59).

[54] URL: http://peak.telecommunity.com/DevCenter/Trellis#model-view-controller-and-the-observer-pattern (visited on 05/12/2014) (cit. on p. 59).

[55] URL: http://knockoutjs.com/ (visited on 05/12/2014) (cit. on p. 59).

[56] URL: http://just-bottom.blogspot.fi/2013/09/a-few-reasons-why-functional-reactive.html (cit. on p. 98).

[57] URL: http://todomvc.com/ (visited on 07/20/2014) (cit. on p. 104).

[58] URL: https://github.com/divipp/lensref/ (visited on 07/20/2014) (cit. on p. 104).

[59] Amann Salvaneschi and Mezini Proksch. *An Empirical Study on Program Comprehension with Reactive Programming*. URL: http://www.guidosalvaneschi.com/attachments/papers/2014_An-Empirical-Study-on-Program-Comprehension-with-Reactive-Programming_pdf.pdf (visited on 07/20/2014) (cit. on p. 103).

[60] Mezini Salvaneschi. *Towards Reactive Programming for Object-oriented Applications*. URL: http://www.guidosalvaneschi.com/attachments/papers/2014_Towards-Reactive-Programming-for-Object-Oriented-Applications_pdf.pdf (visited on 05/20/2014) (cit. on pp. 59, 99, 104).

Glossary

Abstraction Creating classes to simplify aspects of reality using distinctions inherent to the problem . 4

Class A description of the organization and actions shared by one or more similar objects . 4

Combinator A combinator is a function that takes one or more elements and returns a new element on which this combinator or another can be applied again. In this way, combinators can "combine" more complex elements from simpler ones. . 2, 58, 61

Dynamic dispatch The process of selecting which implementation of a polymorphic operation (method or function) to call at runtime . 2, 4

Encapsulation Designing classes and objects to restrict access to the data and behavior by defining a limited set of messages that an object can receive . 4

Functional Reactive Programming (FRP) A functional alternative to event-based programming where certain datatypes represent values over time and dependencies between reactive entities are taken care of automatically . 2

Higher-order function A function that takes one or more functions as input or that outputs a function . 4

Inheritance The data and behavior of one class is included in or used as the basis for another class . 4

Inversion of Control describes a design in which client code receives the flow of control from a framework. With inversion of control, it is the framework that calls into the client code . 6, 59

Message passing An object sends data to another object or asks another object to invoke a method . 4

Method A way to access, set, or manipulate an object's information . 4

Object An individual, identifiable item, either real or abstract, which contains data about itself and the descriptions of its manipulations of the data . 4

Appendix A

Implementation Remarks

The code listings in this thesis have all been implemented in an executable form. The repository containing all the code listings can be found at the following URL: https://github.com/eugenkiss/7guis.

The following versions of the employed tools were used: Java 8, JavaFX 8, Scala 2.11, ScalaFX 8.0.0-R4, ReactFX 1.2, Scala.Rx 0.2.3, Elm 0.12.3. The repository also contains solutions for the case studies in Java7/Swing and Clojure/Seesaw which were implemented before the decision to use JavaFX came about. The repository is structured in a way as to be a notational benchmark for languages and/or libraries/frameworks/toolkits with respect to GUI programming and open for future additions from contributers.

Appendix B

Alternative Approach to Cells

This chapter contains a suggestion by Tomas Mikula how to use the JavaFX binding mechanism to implement the change propagation for the Cells case study. It uses his library EasyBind mainly for the `flatMap` function[1]. In short, the class `Cells` would look something like the following:

```
class Cell {
    BooleanProperty showUserData = new SimpleBooleanProperty(false);
    StringProperty userData = new SimpleStringProperty("");
    ObservableValue<Double> value = EasyBind.map(userData, Parser::parse)
            .flatMap(f -> Bindings.createObjectBinding(() -> f.eval(), f.getReferences()));
    ObjectBinding<String> text = Bindings.when(showUserData)
            .then((ObservableObjectValue<String>) userData)
            .otherwise(EasyBind.map(value, String::valueOf));
}
```

The functions `map` and `flatMap` are from EasyBind. Whenever the value of `userData` changes it is parsed to a formula. For every new formula, a new Binding is created whose dependencies are the referenced cell's values. The function `flatMap` flattens the result, i.e., the return type is `Binding<Double>` instead of `Binding<Binding<Double»`.

[1] https://github.com/TomasMikula/EasyBind

Appendix C

ReactFX Feedback Loops Remarks

This chapter contains remarks by Tomas Mikula about how to prevent feedback loops in cases where the conversion back and forth is not stable, e.g. `fToC(cToF(v)) != v` for the TEMPERATURE case study. It is possible to employ interceptable event streams and `guardedBy`. It complicates the code slightly:

```
val iCelsius = celsius.text.interceptable
val iFahrenheit = fahrenheit.text.interceptable

celsius.text |= iFahrenheit.guardedBy(iCelsius::mute)
        .filter(v => isNumeric(v)).map(v => fToC(v))
fahrenheit.text |= iCelsius.guardedBy(iFahrenheit::mute)
        .filter(v => isNumeric(v)).map(v => cToF(v))
```

This is what happens when the value of fahrenheit text field is changed:

```
fahrenheit.text emits
iFahrenheit emits the same value
iFahernheit.guardedBy(...) emits the same value like this:
    iCelsius is muted first
    value is emitted
        celsius text is updated
        celsius.text emits (but not iCelsius, which is muted)
    iCelsius is unmuted
```

Appendix D

Typical Use of Modules in Elm

The following is an exemplary module structure in Elm given by its author Evan Czaplicki. Let us suppose we want to provide a widget to clients of our widget library. We would subdivide our widget in four modules as follows:

```
-- WidgetModel.elm
module Model (initialize) where
import UnderlyingModel (..)
initialize : Int -> ... -> State
noop : Action
-- WidgetUnderlyingModel.elm
module UnderlyingModel where
type State = { count:Int, ... }
data Action = ...
-- WidgetUpdate.elm
module Update (step) where
import UnderlyingModel (..)
step : Action -> State -> State
-- WidgetView.elm
module View (view) where
import UnderlyingModel (..)
view : Handle Action -> State -> Element
```

Note the module `WidgetUnderlyingModel`. This way we use information hiding to hide implementation details from the clients of our library. In this sense "encapsulation" is encouraged but not in the sense that a module should hold state. If a client wanted to use our widget he does not need to know any facts about implementation details.

```
import Update (step)
import Model (initialize)
import View (view)
```

One thing to consider is making sure that the view function takes an input handle as an argument such that it can be wired up later.

Appendix E

Glitches

The following explanation of glitches is inherited from [5].

> Glitches are update inconsistencies that may occur during the propagation of changes. When a computation is run before all its dependent expressions are evaluated, it may result in fresh values being combined with stale values, leading to a glitch.
>
> Consider an example reactive program below:
>
> ```
> var1 = 1
> var2 = var1 * 1
> var3 = var1 + var2
> ```
>
> In this example, the value of the variable `var2` is expected to always be the same as that of `var1`, and that of `var3` to always be twice that of `var1`. Initially when the value of `var1` is 1, the value of `var2` is 1 and `var3` is 2. If the value of `var1` changes to, say 2, the value of `var2` is expected to change to 2 while the value of `var3` is expected to be 4. However, in a naive reactive implementation, changing the value of `var1` to 2 may cause the expression `var1 + var2` to be recomputed before the expression `var1 * 1`. Thus the value of `var3` will momentarily be 3, which is incorrect. Eventually, the expression `var1 * 1` will be recomputed to give a new value to `var2` and therefore the value of `var3` will be recomputed again to reflect the correct value 4.

Note that Glitches are not exclusive to FRP. They can also happen in classical callback code such as in JavaFX[1]. In Scala.Rx glitches are avoided by using a topologically sorted dependency graph which ensures that an expression is evaluated only after all its dependents have been evaluated. As Scala.Rx's dependency graph is dynamic it can be restructured during

[1] A concrete example can be found here: http://tomasmikula.github.io/blog/2014/05/13/how-to-ensure-consistency-of-your-observable-state.html

run-time and therefore some redundant computations are possible[2] (but not inconsistencies). ReactFX provides some options to manually prevent glitches. Elm prevents glitches automatically by making sure that the order of events is maintained (in some circumstances with dummy events). If the paradigm automatically prevents glitches then this is a win of course as error-proneness is reduced.

[2] E.g. when the system notices during propagation that the graph must be recomputed the computed values so far will be recomputed.

Appendix F

Further Code Listings

```
public class TemperatureConverter extends JFrame {
    JTextField celsiusField;
    JTextField fahrenheitField;

    public TemperatureConverter(String name) {
        super(name);
        initGUI();
        initListeners();
    }

    private void initGUI() {
        celsiusField = new JTextField(5);
        fahrenheitField = new JTextField(5);

        Container pane = this.getContentPane();
        pane.setLayout(new FlowLayout());
        pane.add(celsiusField);
        pane.add(new JLabel("Celsius"));
        pane.add(new JLabel("="));
        pane.add(fahrenheitField);
        pane.add(new JLabel("Fahrenheit"));
    }

    private void initListeners() {
        celsiusField.getDocument().addDocumentListener(new DocumentListener() {
            public void insertUpdate(DocumentEvent e) { update(); }
            public void removeUpdate(DocumentEvent e) { update(); }
            public void changedUpdate(DocumentEvent e) { update(); }

            private void update() {
                if (!celsiusField.isFocusOwner() ||
                        !isNumeric(celsiusField.getText())) return;
                double celsius = Double.parseDouble(celsiusField.getText().trim());
                double fahrenheit = celsiusToFahrenheit(celsius);
                fahrenheitField.setText(String.valueOf(Math.round(fahrenheit)));
            }
        });
        fahrenheitField.getDocument().addDocumentListener(new DocumentListener() {
```

```
            public void insertUpdate(DocumentEvent e) { update(); }
            public void removeUpdate(DocumentEvent e) { update(); }
            public void changedUpdate(DocumentEvent e) { update(); }

            private void update() {
                if (!fahrenheitField.isFocusOwner() ||
                        !isNumeric(fahrenheitField.getText())) return;
                double fahrenheit = Double.parseDouble(fahrenheitField.getText().trim());
                double celsius = fahrenheitToCelsius(fahrenheit);
                celsiusField.setText(String.valueOf(Math.round(celsius)));
            }
        });
    }
    // The rest is omitted for space reasons.
}
```

Listing 15: Temperature Converter in Java7/Swing.

```
(defn convert-panel []
  (let [celsius    (text :columns 5)
        fahrenheit (text :columns 5)]
    (b/bind
      celsius
      (b/filter #(and (.isFocusOwner celsius) (numeric? %)))
      (b/transform #(-> % parse-temp c-to-f display-temp))
      fahrenheit)
    (b/bind
      fahrenheit
      (b/filter #(and (.isFocusOwner fahrenheit) (numeric? %)))
      (b/transform #(-> % parse-temp f-to-c display-temp))
      celsius)
    (flow-panel
      :items [celsius "Celsius" "=" fahrenheit "Fahrenheit"])))

(defn -main [& args]
  (invoke-later
    (-> (frame :title "Temperature Converter" :content (convert-panel) :on-close :exit)
     pack!
     show!)))
```

Listing 16: Temperature Converter in Clojure/Seesaw.

```
implicit class EventStreamExtensions[T](val stream: EventStream[T]) {
  def /[U](denom: EventStream[U])
          (implicit ev1: T => Number,
                    ev2: U => Number): EventStream[Double] =
    combine(stream, denom)
        .map(case (a, b) => ev1(a).doubleValue / ev2(b).doubleValue)
}
```

Listing 17: A sketch of an improved integration of ReactFX in terms of binary operators by Tomas Mikula. With it, it would be possible to write `progress.progress |= elapsed / slider.value` for the Timer case study. In practice, it was not managed to make the Scala compiler accept this integration reasonably.

```
this.mousePresses
    .hook(e => popup.hide())
    .filter(e => e.isPopupTrigger)
    .map(e => (e, getNearestCircleAt(e.x, e.y)))
    .filter{case (e, c) => c != null}
    .hook{case (e, c) => popup.show(this, e.screenX, e.screenY)}
    .emitOn(diameter.actions)
    .subscribe{case (e, c) => {
        popup.hide()
        showDialog(c)
    }}
```

Listing 18: An alternative, more succinct way to express parts of Circle Drawer with ReactFX by Tomas Mikula. Note that `getNearestCircleAt` is only called once.

List of Code Listings