Solving the Overproduction Problem in Reactive Programming using Feedback Control

Master Thesis

Richard van Heest

December, 2016

Solving the Overproduction Problem in Reactive Programming using Feedback Control

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Richard van Heest
born in Middelharnis, The Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, The Netherlands
www.ewi.tudelft.nl

Solving the Overproduction Problem in Reactive Programming using Feedback Control

Author: Richard van Heest
Student id: 4086570

Abstract

In an interactive program the consumer requests data from the producer and therefore has control over the speed at which data is consumed. This is opposite to a reactive program, where the producer emits data to the consumer at its own pace. This forms a problem if the producer emits more data than the consumer can deal with. Solving this '*overproduction*' problem requires a good understanding of the nature of a reactive program, as well as the various behaviors it can have. Several solutions have been proposed and implemented, though they turn out to not work under all circumstances. We classify these solutions based on the type of reactive programs they can be applied to. We further develop a new solution for the case in which the producer can actually be controlled. Our solution involves the notion of feedback control, which is a technique that is not well-known in computer science. For this technique to be applied, we develop an API to compose and execute feedback systems in production level systems. Based on this API we develop our solution to the overproduction problem.

Acknowledgment

I would like to thank Erik Meijer, my supervisor during this thesis project, for all his support, enthusiasm and hacker mentality. *Erik, it was an honor to receive an email in May 2014, asking whether I was interested in doing my master thesis with you. From the beginning it was clear to both of us that Reactive Programming and Rx were a shared interest. It was great working with you on RxMobile, which has turned out to be very useful for this thesis. You really inspired me to have a hacker mentality, always keep trying to get things working and to never give up. I sincerely hope the completion of this thesis will not be the* `onCompleted` *event on the stream of our collaborations, but instead will just be the* `onNext` *event of a finished project on a much longer stream!*

I also want to thank Georgios Gousios and Arie van Deursen for taking their time to be part of my thesis committee and providing me with great feedback.

Furthermore, I would like to thank my fellow students and friends at TU Delft, Eddy Bertoluzzo, Georgi Khomeriki, Mircea Voda, Mike de Waard and Lars Willems, who were doing their master theses in parallel with me. Thanks for the great time we had together at EWI-HB 08.250 and the many discussions at the whiteboard and at Hangouts/Skype!

A special thanks to Michel van Heest, my brother, for creating all the diagrams in this thesis.

Finally, I would like to thank my family, friends and colleagues for their support and interest in my thesis work, even though most of them still don't understand a word of what will follow in the rest of this thesis!

Richard van Heest
Middelharnis, The Netherlands
December, 2016

Contents

List of Figures

Chapter 0

Introduction

Reactive programming is a paradigm in which the program observes events that occur in its environment and reacts to these events as they occur. This is in contrast to the more familiar interactive program flow in which a program *requests* some form of data and only continues the program flow once this data is received. Instead of pulling data in the usual interactive way, a reactive program has data (or events) being pushed at it, on which it responds accordingly. Some of the most common usages of reactive programming can be found in user interface interaction (mouse moves, key events or button presses), network communication, database queries, clocks and timers [32].

In an interactive program, the *consumer* is in charge of requesting the data. The *producer* only has to obey the commands from the consumer and return the requested data. In reactive programs this is the exact opposite: the producer is in charge and sends the data *at its own pace* [20], whereas the consumer has to react to the data it receives.

This shift in roles between producer and consumer poses an interesting problem: "*What happens when the consumer cannot keep up with the amount of data that is sent by the producer?*". In other words, when a consumer has to deal with an overproducing source, how can it handle the excessive amount of data? A naive solution is to buffer the remaining elements for later processing. This, however, has the risk of the program running out of memory. Several other solutions with different policies have already been proposed and implemented in order to solve this problem [3, 6, 8]. We find, however, that some of these work well under certain circumstances but are not suitable for all kinds of reactive programs in general. Other solutions change the contract of reactive programming and put the consumer back into command, but, surprisingly enough, still call it 'reactive'.

This poses the question whether the concepts behind reactive programming have been generalized too much and whether distinctions between different kinds of reactive programs can lead to a clearer view of what solution works best under which conditions.

We propose a new solution to the overproduction problem, which makes use of control theory and feedback control systems. This solution can be used as a replacement to solutions that put the consumer back in charge.

Feedback control is a technique that is mainly used in mechanical and electrical engineering, but is generally overlooked in computer science. Feedback control can be applied by continuously measuring a system's property, comparing it to a desired value and alter the system's input based on the error between the desired and measured values. The goal is to ultimately bring the measured property as close to the desired value as possible and keep it this way despite external changes that try to bring the system out of balance [28].

Although well suited in several use cases, control theory is generally overlooked in computer science and software engineering. To the best of our knowledge, there are no well-written libraries that allow software engineers to construct and run feedback systems in a production level system. As we will use feedback control in our solution for the overproduction problem, we will study the composition of feedback systems and present a concise but powerful library for this purpose that is based on the concepts of functional and reactive programming. Using this library, we will then propose our solution to the overproduction problem.

0.1 Research Questions

This thesis answers a number of questions that are related to reactive programming, overproducing sources in a reactive context and feedback control, which are listed and briefly introduced in this section.

In which ways can a reactive program already be controlled to prevent overproduction?

There are multiple solutions for controlling overproduction in the context of reactive programming. To understand these solutions, we first need to identify what various types of reactive programs exist and consider their mutual similarities and differences. Based on this we can identify which existing solution for controlling overproduction will or will not work for each particular type of reactive program. This analysis can further be used to create a better understanding of which type of reactive program is suited for the solution proposed in this thesis.

How can we implement a *reactive* feedback system that is composed of smaller parts?

As the solution to overproduction that is presented in this thesis will make use of control theory and feedback control, it is important to develop an understanding of these techniques and create the tools necessary to construct feedback systems in an easy way. This is particularly important since (to the best of our knowledge) no libraries or APIs exist that allow for composing and running feedback systems. We explore how a feedback system can be seen as a reactive program and implement a library on top of existing reactive programming API's.

How can the overproduction problem be reduced to a feedback control problem?

In order to solve the problem of overproduction using feedback control, it is necessary to get out of the context of reactive programming and abstract this into more formal problems that can be solved using the principles control theory. Based on the feedback system that solves this control problem, a mapping can be created to the original problem of controlling overproduction in a reactive program.

0.2 Outline

In the next chapter we will continue this thesis report with a short discussion on the definitions of interactive and reactive programming as it is defined in literature (Chapter 1). As we do not assume any prior knowledge of the reader in reactive programming, this is followed with an extensive introduction to a widely used reactive programming API whose concepts have been implemented in many languages. We will discuss the basic principles of this API as well as it's derivation from and relationships with interactive programming interfaces. Once these foundations are laid out, we will give an overview of the different ways in which it tries to overcome overproduction problems. It is important to first gain a good understanding of reactive programming in general and this API especially, since we will build on top of these in the rest of this thesis.

Different circumstances require different approaches to overproduction, as we already hypothesized above. In Chapter 2 we will explore and categorize these circumstances and describe how overproduction is currently best dealt with in each of these situations.

As we will use control theory in our solution to the problem of overproduction, and given that this topic is not well known in computer science, we will briefly introduce this topic in Chapter 3. We also provide an extended example of a simple way to apply feedback control.

To the best of our knowledge there does not exist a proper API, library or framework for building and running feedback control systems in production level systems. In Chapter 4 we derive a simple API for such feedback control systems. This API is based on the concepts of functional and reactive programming and has its basis in the solid foundations of theoretical computer science and category theory. We also return to the example from Chapter 3 and rewrite this application such that it uses our newly designed API.

Chapter 5 combines the analysis about overproduction and the theory of feedback control, and with that proposes a new way to deal with overproduction. We will transform this problem into a more general control problem, discuss the various aspects of the resulting feedback system and integrate it in such a way that it can be used as part of a reactive API like RxJava or RxMobile.

Chapter 1

Programming in a reactive way

In order to have a good understanding of the true meaning and problem of overproduction, it is important to first define what *reactive programming* is, what its relation to other programming patterns is and how this paradigm is implemented in a library like *Reactive Extensions*, which will be used throughout this thesis.

In order to understand these fundamental principles, this chapter starts off with defining what reactive programming is, as it is described in the literature. Since this paradigm forms the basis of this whole thesis, it is important to establish the exact notion of reactiveness first.

Then the Reactive Extensions API is discussed. It is the context in which overproduction occurs as well as the basis of the tools we will be using to solve this problem. Since this API is not yet considered to be common knowledge and since not much literature is based on it, we will start from the very basics and discuss all the concepts necessary for this thesis.

At the end of this chapter the overproduction problem is introduces, as well as related work that already provides solutions to this problem.

1.1 Reactive Programming

Currently one of the most difficult problems in computer science is handling big amounts of data. No longer are applications bound to the closed world of a single machine and a relational database. Applications these days have access to the whole World Wide Web, exist of large clusters of machines, work with data ranged from SQL-style relational databases to key-value pointer based databases, as well as binary data such as images, audio and video. Also the speed in which data is handled varies from 'once a month' to 'every millisecond'.

These changes in how applications need to perform require new ways of handling data. No longer is it feasible to load a whole database table into memory for further processing, nor can we permit ourself to wait for all data to be downloaded before we start processing it. Instead we require systems and concepts that are able to handle data right as it gets available to the application, without further delay, preferably in an asynchronous way and without blocking other processes or waiting for all data to have arrived. [32]

An interesting part of the solution to these problems is Reactive Programming. This term is described by Gérard Berry in "*Real time programming: special purpose or general purpose languages*" [20] as he makes a distinction between *interactive* and *reactive* programs:

> "*Interactive programs* interact at their own speed with users or with other programs; from a user point of view, a time-sharing system is interactive. *Reactive programs* also maintain a continuous interaction with their environment, but **at a speed which is determined by the environment**, not by the program itself."

Reactive programs 'observe' events that occur in their environment and react to them as specified by the program. These events can vary from large amounts of data coming in over a network connection or from a database, to mouse moves or other kinds of UI events and from low-latency sensor streams to high-latency calls to web services.

Also notice that Berry explicitly emphasize that that reactive programs run at a speed which is determined by the environment. This means that the producer (which is part of the environment) is in charge of sending data to the consumer. Therefore the consumer (the program) cannot *ask* for new data, it can only *react* to the data that has been sent by the producer. This relationship between producer and consumer is often referred to as *push based behavior.*

A classic example of push based behavior is a mouse pointer moving over the screen. Every time the pointer is moved, it will *push* its new coordinates to the screen, for it to drawn in the new position. From its point of view, the screen *reacts* to the new coordinates being received by the mouse pointer.

This push based behavior is in contrast with the relation between producer and consumer in an interactive program. Here, according to Berry, the consumer (program) interacts on its own speed with its environment. The consumer will determine the speed at which data is transmitted from the outside by continuously asking for the next bit of data. After this is received, the consumer processes the data and then asks for the next piece. This kind of interaction between consumer and producer is often referred to as *pull based behavior*.

One example of a simple interactive program is a foreach-loop iterating over a collection of elements. As long as there are more elements in the collection, it will ask for the next one, process it according to the loop's body and then ask for the next element.

1.2 Reactive Extensions

There have been many attempts to fit the philosophy of reactive programming into frameworks, APIs and even languages [1, 4, 8, 10, 17, 37]. In this section, we will discuss some of the features of one of these libraries, namely Reactive Extensions (a.k.a. Rx). This project started at Microsoft with an implementation in C# [31] (Rx.Net), was later ported to Java, Scala, Groovy, Kotlin, JavaScript, Swift and many other languages by the open source community [10]. It is currently the standard library for programming in a reactive way.

Unfortunately, these various translations have each been evolving in their own way, deviating from both the original implementation as well as each other. There are obvious minor changes such as operator names changing to conform particular language standards, but also behavior in various corner cases changed. Most remarkable however is that some implementations are not even purely 'reactive' anymore [34]. Given these deviations from the original paradigm and the state of complexity of these implementations, we decided to use a reference implementation of the original Rx that has recently been written in Scala by Erik Meijer et al. called RxMobile [17], with the purpose of creating a light-weight implementation for mobile app development. The following discussion and derivation of the API will however apply to both Reactive Extensions and RxMobile and in this section we will therefore refer to both of them as 'Rx'.

1.2.1 Core components

Rx is a library for composing asynchronous and event based (reactive) programs by using observable sequences [11]. The core of Rx consists of two interfaces: `Observable` and `Observer`. The latter can subscribe and react to the events that are emitted by the former. An `Observable` can emit zero or more events (called *onNext*) and has the possibility to terminate with an *onCompleted* or *onError* event. After either one of these terminal events is emitted, no more events can follow. Therefore the emission protocol can be summarized by the following regular expression: `onNext* (onError | onCompleted)?` [18]. When an `Observer` subscribes to an `Observable`, it will return a `Subscription`. With this object reference, one can later unsubscribe from the `Observable` and clean up potential resources.

Listing 1.1 shows these basic concepts of the `Observable`, `Observer` and `Subscription` translated in Scala. Notice that here `Subscription` is a superclass of `Observer`. Therefore there is no need for the `Observable` to return a `Subscription` when an `Observer` subscribes to it. It will however return a `Subscription` when another variant of `subscribe` is used, where for example a lambda expression is expected instead.

Creating an `Observable` is done by the `Observable.create(Observer ⇒ Unit): Observable` method, that takes a lambda expression of type `Observer ⇒ Unit` and returns an `Observable`. The input lambda is then used in the implementation of `subscribe`, when a *real* `Observer` is provided. The `Observer` can be created by supplying it three lambda expressions, one for each kind of event.

Listing 1.1: Observable, Observer and Subscription

```scala
trait Observable[T] {
    def subscribe(observer: Observer[T]): Unit
    def subscribe(onNext: T => Unit): Subscription
    // other variants of subscribe
}
object Observable {
  def create[T](create: Observer[T] => Unit) = new Observable[T] {
    override def subscribe(observer: Observer[T]) = create(observer)
  }
  // other ways to create an Observable
}

trait Observer[T] extends Subscription {
    def onNext(t: T): Unit
    def onError(e: Throwable): Unit
    def onCompleted(): Unit
}

trait Subscription {
    def isUnsubscribed(): Boolean
    def unsubscribe(): Unit
}
```

Listing 1.2 provides a simple example of how both an `Observable` and `Observer` are created and used in practice. Here the function in `Observable.create` causes the `Observable` to emit three values and complete afterwards. Notice that these are only emitted after line 12 is executed, when the `Observer` is subscribed to the `Observable`. If no one will subscribe to the `Observable`, the lambda expression in `Observable.create` will never be executed and hence none of these value will ever been emitted!

Listing 1.2: Creating and subscribing to an `Observable`

```scala
 1  val xs: Observable[Int] = Observable.create((obv: Observer[Int]) => {
 2      obv.onNext(1)
 3      obv.onNext(2)
 4      obv.onNext(3)
 5      obv.onCompleted()
 6  })
 7  val observer: Observer[Int] = Observer(
 8      (x: Int) => print(x + " "),
 9      (e: Throwable) => print(e),
10      () => print("completed"))
11
12  xs.subscribe(observer)
13
14  // result: 1 2 3 completed
```

Using `Observable.create` is a very powerful tool to create an `Observable`. Many other methods can be derived from it. For example, the `Observable` in Listing 1.2 is often written as `Observable.apply(1, 2, 3)`[1]. This way of writing is not only more concise and conveys what the true meaning of this expression is in a better way, but it is also exactly the same, since `Observable.apply` is implemented in terms of `Observable.create`. In fact, all methods and operators that are defined on `Observable` are implemented using `Observable.create`!

1.2.2 Derivation of `Observable` and `Observer`

In 1994, the book '*Design Patterns: Elements of Reusable Object-Oriented Software*' by the *Gang of Four* was published [23]. This book explored the capabilities and pitfalls of object oriented programming and contained an overview of 23 classical software design patterns. Also, the book described the relationships between these 23 design patterns.

One of these design patterns is called the *Observer* pattern and forms the basis of the `Observable` and `Observer` interfaces described in the previous section. Even though the Gang of Four did identify a lot

[1]In Scala this can be shortened to `Observable(1, 2, 3)`. Explicitly writing `.apply` is only done for later referral.

of relations between the different design patterns, it failed to identify any relation between the Observer pattern and any other pattern, except for the Mediator pattern.

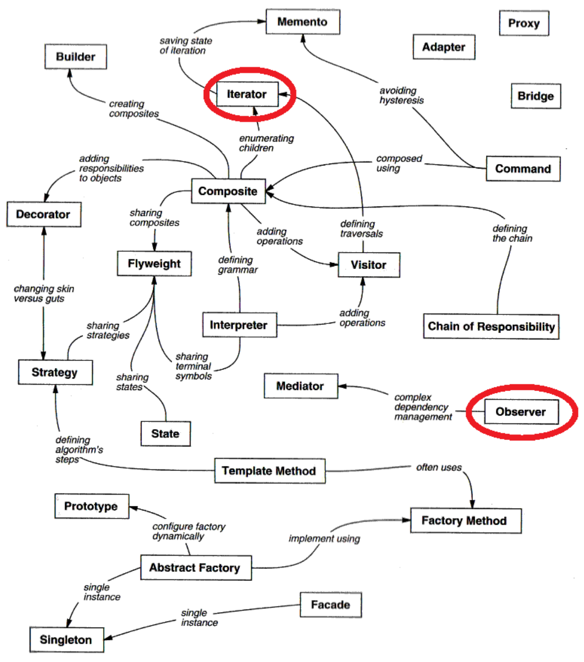


Figure 1.1: Relations between design patterns

In 2010, a short paper called '*Subject/Observer is Dual to Iterator*' [31] was published by Erik Meijer, describing a mathematical relationship between the Observer pattern and the Iterator pattern based on categorical duality that had not been reported upon earlier. The paper shows that instances of the Observer pattern can be viewed as push-based collections, rather than the pull-based collections that result from the Iterator pattern. For later parts of this thesis, it is important to understand the mathematical basis of this relationship between the `Observable` and `Observer` interfaces in Rx and the `IEnumerable` and `IEnumerator` interfaces in the Iterator pattern[2] (see Listing 1.3).

In most common languages `IEnumerable` forms the basis of the Collections API. It has only one method `getEnumerator` that returns the `IEnumerator` to iterate over the elements in the collection. The `IEnumerator` interface on the other hand contains two methods to be implemented: `moveNext` and `current`. The former performs a side effect by moving a pointer to the next element in the iteration and then returns a `Boolean` to indicate whether or not there was a next element. The latter is a pure function that just returns the element the pointer is currently pointing to. Notice that the `moveNext` method can throw an exception rather than returning `false` in case an error occurs.

Besides providing these two methods, `IEnumerator` in Listing 1.3 also extends the `IDisposable` interface. This interface is meant to signal to the `IEnumerable` that no more elements will be pulled and that it can 'start collaborating' with the garbage collector to clean up resources. The real meaning of `IEnumerator` extending `IDisposable` however, is that `getEnumerator` not only returns an `IEnumerator`, but also returns something that is disposable. The `IDisposable` interface is therefore not really part of `IEnumerator` but rather a part of what `getEnumerator` returns [35]. For now we will consider `IDisposable` to be a silent bystander that is will be ignored in the derivation.

Listing 1.3: `IEnumerable` and `IEnumerator` interfaces

```
trait IEnumerable[T] {
    def getEnumerator(): IEnumerator[T]
}
trait IEnumerator[T] extends IDisposable {
    def moveNext(): Boolean // throws Exception
    def current: T
}
trait IDisplosable {
    def dispose(): Unit
}
```

[2]For the purpose of the upcoming derivation we have chosen the C# naming conventions of the Iterator pattern. In many other programming languages these interfaces are respectively known as `Iterable` and `Iterator`.

These two interfaces together form the basis of all pull-based or interactive collections as described in Section 1.1. The user asks for the next element and will get one in case a next element can be produced. In the following we will transform these interfaces for pull-based collections into interfaces for push-based or reactive collection, where the user subscribes to a collection and receives data once it is produced. This derivation, as well as its conclusion that interactive and reactive collections are each other's dual, is based on some categorical transformations and are discussed in several papers, as well as several keynotes and Channel9 video's [31, 32, 33, 35]. This derivation, as well as some of the intermediate steps are important for later parts of this thesis.

The first step in this derivation is to rewrite the two methods in the `IEnumerator` interface into a single method `getNext()`. Using the categorical *coproduct* [38] we can combine these two methods and determine its type signature: `getNext()` can either fail with an exception or succeed with either zero or one elements, resulting in the type signature `getNext(): Try[Option[T]]`. The new, intermediate, set of interfaces is shown in Listing 1.4.

Listing 1.4: `IEnumerator` interface after applying coproduct

```
trait IEnumerable[T] {
    def getEnumerator(): IEnumerator[T]
}
trait IEnumerator[T] {
    def getNext(): Try[Option[T]]
}
```

Since both interfaces now only have one single method, and given that the only purpose of `IEnumerable` is to produce an `IEnumerator`, they can be written as a single lambda expression. An `IEnumerable` can be written as:

$$\texttt{()} \Rightarrow \texttt{(()} \Rightarrow \texttt{Try[Option[T]])} \tag{1.1}$$

Notice that applying `Unit` to the outer lambda yields another lambda expression, which corresponds to the type signature of `getNext` in Listing 1.4: `() ⇒ Try[Option[T]]`.

The next step in this transformation is to dualize [38] lambda expression 1.1. A very informal way of describing duality is to flip all the arrows and rewrite the lambda expression. For example, the duality of $f :: A \to B$ is $\bar{f} :: A \leftarrow B \equiv B \to A$. In the same way, we can apply this to lambda expression 1.1, resulting in

$$\texttt{(Try[Option[T]]} \Rightarrow \texttt{())} \Rightarrow \texttt{()} \tag{1.2}$$

This lambda expression takes a lambda `Try[Option[T]] ⇒ Unit`, and returns `Unit`.

We can now put this lambda expression back into context by splitting it into two interfaces. The inner lambda `Try[Option[T]] ⇒ ()` can be rewritten to an interface called `Observer`, which has one method `onNext(t: Try[Option[T]]): Unit`. This method can then be further rewritten into three separate methods by expanding the `Try[Option[T]]` type: `onNext(t: T): Unit`, `onError(e: Throwable): Unit` and `onCompleted(): Unit`. The outer lambda on the other hand translates to an interface called `Observable`, which has one method `subscribe(obv: Observer[T]): Unit`. Notice how these interfaces are completely identical to the ones presented in Listing 1.1.

So far, the presence of `IDisposable` has been ignored in this whole derivation. The reason for that is that this interface is considered to be a *second* thing that is returned by the `IEnumerable`, rather than a supertype of `IEnumerator`. What therefore basically happened in the derivation is that we only dualized the enumerator*ness* and left the disposable*ness* out of the dualization process [35]. Therefore the dualized `IEnumerator`, now called `Observer`, still extends from `IDisposable`, even though this only means that we pass *two* arguments to the `Observable.subscribe(obv: Observer)`. Just as the `IDisposable` was meant to signal to the `IEnumerable` that no more elements will be polled and that it can clean up its resources, now `IDisposable` signals to the `Observable` that it should stop sending data to the `Observer`. Finally, `IDisposable` is renamed to `Subscription` and its method `dispose` is split into two methods `unsubscribe` and `isUnsubscribed`.

This derivation shows that interactive, pull-based collections are the mathematical dual of reactive, push-based collections. The `Observable` and `Observer` interfaces can directly be derived from the `IEnumerable` and `IEnumerator` interfaces. Both sets of interfaces can therefore be considered to be collections. In other words: streaming data behaves exactly the same way as regular collections, such as arrays, lists and sets, except for them being push-based rather than pull-based [31, 32]. In the world of push-based collections one *subscribes* to the stream in order to *react* to the next element that is being send, whereas one *asks* for the next element in a pull-based scenario.

1.2.3 `Observable` as a monad

As described in the previous section, `Observable` can also be written as lambda expression 1.2. A better look at this expression reveals that `Observable` is actually a special instance of the *continuation monad*, which has the following type:

$$\texttt{(S} \Rightarrow \texttt{R)} \Rightarrow \texttt{R} \tag{1.3}$$

In the `Observable` lambda expression, `S` is equal to `Try[Option[T]]` and `R` is equal to `()` or `Unit`.

Given that `Observable` is just a continuation monad and hence an instance of `monad`, it automatically inherits the two operators that are defined on all monads: `return` and `(>>=)` ('bind'). In the Rx implementation, these operators are present as well. The `return` creates an `Observable` from a `Try[Option[T]]`, meaning that it accepts either an error, or an empty value, or a non-empty value. Therefore `return` is split into three operators `apply(t: T)`, `error(e: Throwable)` and `empty()`. Since an `Observable` can have multiple values, `apply` is overloaded to have more than one value. This overload was already shown in Section 1.2.1. The `(>>=)` operator is renamed to `flatMap` and also splits the `Try[Option[T]]` parameter into three separate parameters [13]. Besides that, since the `T ⇒ Observable[S]` parameter is used most frequently, the `flatMap` operator is overloaded with only this parameter.

A simple example of using these monadic operators in Rx is shown in Listing 1.5. On line 1 the overloaded `apply` is called, which lifts four values into the `Observable`. The `flatMap` operator on line 2 doubles the number of elements by creating an `Observable` that emits the value as well as the square of the value.

Listing 1.5: Monad operators in Rx

```
1 Observable(1, 2, 3, 4)
2     .flatMap(x ⇒ Observable(x, x * x))
3     .subscribe(x ⇒ print(x + " "))
4
5 // result: 1 1 2 4 3 9 4 16
```

1.2.4 Operators

In Section 1.2.2 we concluded that both the Iterator pattern and the Observer pattern are collections, only separated by the difference between push-based and pull-based behavior. All other rules on collections do however apply to both of them. In regular pull-based collections many operators are defined to manipulate, transform, filter, fold or group elements. These operators can therefore also be applied to push-based collections. One of them, `flatMap` was already shown in the previous section. However, rather than iterating over the pull-based collection and applying a transformation to each element, these operators *react* to data being emitted by applying their particular transformation or side effect and passing the (transformed) data down to either a potential next operator or the `subscribe` method.

The Rx implementations of the `Observable` interface provide a wide variety of operators that apply all sorts of transformations to a data stream [13]. All operators are defined on `Observable` and will also return an `Observable`, making the API highly compositional. In order to understand how these operators work, we will look at some basic examples. Other, more advanced operators will be discussed in Section 1.3.1.

Filter To select only those elements that satisfy a certain predicate, the operator `filter(p: T ⇒ Boolean): Observable[T]` is used. Every time an element is received by this operator, the predicate `p` will be applied. If the element satisfies the predicate, it is passed downstream; otherwise the element will be discarded. Listing 1.6 shows in Line 2 how to select the odd numbers in a stream of integers by supplying a predicate.

Map To transform one stream of data into another, the `map(f: T ⇒ S): Observable[S]` is used. Each time an element (which is of type `T`) is received by this operator, the function `f` is applied to this element, yielding a new element of type `S`. This new element is then passed to down the stream. In Listing 1.6 the `map` operator is first applied in line 3 to the stream of filtered elements with a function that doubles the input.

Scan Most operators do not allow for any form of internal state. They do not keep track of previous elements. An operator that can take the previous elements into account is `scan(seed: S)(acc: (S, T) => S): Observable[S]`. To this operator first of all a seed is supplied, which is the internal state of the operator before any value is received. Once an element is received, it will apply its internal state, together with that element to the accumulator function `acc` and produce an element to be emitted. This emitted value is also the new internal state of the operator. Listing 1.6 has a `scan` operator in line 4 that takes the sum of all integers it receives and uses a `seed = 0`.

Drop The `scan` operator is often used together with `drop(n: Int): Observable[T]`, which discards the first `n` elements and forwards all elements after that. The combination with the `scan` operator is used to prevent the seed value from being emitted further downstream, as is shown in Listing 1.6 line 5.

Take Whereas `drop` discards the first `n` elements, `take(n: Int): Observable[T]` is used to only propagate the first `n` elements and discard all elements that come after that. In practice this means that the stream is terminated early with a call to `Observer.onCompleted()`. Listing 1.6 shows how `take` is used to only propagate the first and the second element and discard the third.

Listing 1.6: Operators on `Observable`

```
1  Observable(1, 2, 3, 4, 5)              // emits:    1, 2, 3, 4,  5
2      .filter(x ⇒ x % 2 == 1)            // emits:    1,    3,     5
3      .map(x ⇒ x * 2)                    // emits:    2,    6,    10
4      .scan(0)((sum, x) ⇒ sum + x)       // emits: 0, 2,    8,    18
5      .drop(1)                           // emits:    2,    8,    18
6      .take(2)                           // emits:    2,    8
7      .subscribe(x ⇒ println(x))
```

Just as the interactive collections, Rx has defined its operators in a way that composition of operators is very easy. In this way, simple operators can be chained in order to create the complex behavior that is often desired. There are many more operators defined on `Observable`, which are not mentioned in this section. For a full overview, we refer to the documentation on the Rx websites [10, 12, 13].

1.2.5 Hot and cold streams

There are many kinds of observable streams that can all be implemented using Rx. For example, a clock or a timer is basically a stream of 'ticks' that emits an element every time unit and therefore has a constant speed. A stream of keyboard events on the other hand emits an element every time a key is pressed and therefore most likely has a very irregular speed. A data stream can also be the result of a database query or a network call. In these instances it might take a certain amount of time before the first result is emitted, but every other result is received almost immediately after the first result appeared.

Some of these data streams, like the database query, are finite and will at a certain time in the future call `onCompleted`. Others, like the clock, will keep producing next elements forever, be it at a regular pace or quite irregular, like the keyboard. This kind of stream will never call `onCompleted`, but still may terminate with an error by calling `onError`.

One other difference between certain streams is what happens when one subscribes multiple times to the same stream. Clocks or keyboard events, like broadcasters, emit values whether or not anyone is subscribed. If no one is subscribed, the events are still produced, but are immediately discarded. On the other hand, if multiple observers subscribe to the same stream, they will all receive the same events. This kind of stream is referred to as a *hot* stream.

Some streams, like the `Observable(1, 2, 3, 4)` in Sections 1.2.1 and 1.2.3 or the database query, are not considered to be broadcasters. This kind of stream will create a new instance of itself every time an `Observer` subscribes to it. A second subscriber therefore receives the same result as a first subscriber, even though the second subscribes much later than the first one. This kind of stream is referred to as a *cold* stream.

Notice that even though these differences do exist, they are not reflected in they type of the `Observable`. It is therefore always good to be careful with these distinctions and not to make any assumptions on streams being hot, cold, finite, infinite or error prone.

1.2.6 Subjects

A `Subject` can be viewed as a bridge between the `Observer` and the `Observable`. It can be subscribed to like an `Observable`, but can also observe another stream like an `Observer`. This is a very powerful tool that is often used as a starting point for a stream. Every time a certain event happens outside the context of the `Subject`, its `Observer` part can be called using the three methods. It will then process these events in its `Observable` part and propagate them down the stream.

A `Subject` can also be used to convert a cold stream into a hot stream. For this, a cold `Observable` is subscribed to the `Subject`. Because of this subscription, the cold `Observable` will be triggered to start emitting its events. The observable part of the `Subject` then becomes a hot `Observable`.

A special instance of `Subject` is the `BehaviorSubject`, which behaves like a normal `Subject` but additionally emits its most recent value (or a seed or default value if none has been emitted yet) immediately after an `Observer` is subscribed to it. This is often used in user interface components like a text field to signal a certain initial state.

1.2.7 Concurrency

One final feature that makes Rx the '*library for composing asynchronous and event based (reactive) programs*' is the way it handles concurrency. In most cases this is the hardest job for any developer and is often the cause of many bugs, deadlocks and race conditions. In Rx, however, this is fully abstracted into two simple operators and one extra interface. With this the API allows for switching from one thread to the other without the developer having to write a lot of code and without introducing inevitable concurrency bugs.

The first operator, called `observeOn(scheduler: Scheduler)`, propagates its received data in a Rx sequence to another thread. All mechanics of switching between threads is abstracted by the `Scheduler`, which takes workloads and schedules them on the thread it represents. By providing the `observeOn` a `Scheduler`, every bit of handling concurrency is done. This is shown in Listing 1.7, where an `Observable` emits the numbers 1 to 4 on the application's main thread, and where every number is doubled on that same thread. The `observeOn` operator, provided with a `NewThreadScheduler`, switches and propagates all data it receives to that `Scheduler`. All operations that are performed by both operators and `subscribe` methods after an `observeOn` is declared are switched to associated `Scheduler`.

One common use case of this operator is found in developing applications with a user interface (UI). When a textfield exposes an `Observable` of keyboard events, we may safely assume that it runs on the UI thread. The content of this textfield is transformed into a query, which is send of to a database, where it will collect and return a certain result set. While the database is processing the query, we do not want to block the UI thread, but rather keep that thread available for other, UI related processes. In this case it very useful to execute the database query on a different thread. For this purpose, RxJava provides a special IO `Scheduler` that is completely optimized for its cause. Once the result of the database query is received, a UI update needs to happen for the result to be shown. This can only be done from within the UI thread, hence the `Observable` needs to be switched back to that `Scheduler` using the `observeOn` operator.

Listing 1.7: `observeOn` in a Rx sequence

```
Observable(1, 2, 3, 4)               // main thread
    .map(i ⇒ 2 * i)                  // main thread
    .observeOn(new NewThreadScheduler)
    .map(i ⇒ i / 2)                  // thread 2
    .subscribe(i ⇒ print(i + " "))  // thread 2
```

Something that is not possible with the `observeOn` operator is to schedule an `Observable` *source* a different thread. For this purpose the `subscribeOn(scheduler: Scheduler)` is also part of the Rx API. This operator schedulers creation of all values within the source on the associated scheduler and also performs all operations on that `Observable` on the same `Scheduler` until a potential other `Scheduler` is encountered in the sequence of operator. This is described in Listing 1.8, where the creation of the values within `Observable.apply` is scheduled on a new thread. Compare this to Listing 1.7, where this was done on the application's main thread. Also the first `map` operation in Listing 1.8 is performed on that same thread. When on line 4 the `observeOn` is encountered, the received data is rescheduled to a second thread.

Listing 1.8: `subscribeOn` in a Rx sequence

```
1 Observable(1, 2, 3, 4)               // thread 1
2     .subscribeOn(new NewThreadScheduler)
3     .map(i ⇒ 2 * i)                  // thread 1
4     .observeOn(new NewThreadScheduler)
5     .map(i ⇒ i / 2)                  // thread 2
6     .subscribe(i ⇒ print(i + " "))  // thread 2
```

More complex configurations of `subscribeOn` and `observeOn` can be made, as shown for example in [15]. Although very interesting, we consider these not relevant for the scope of this thesis. The main point here is that a stream can operate on different threads and that introducing concurrency is as easy as adding one line of code.

1.3 Fast producers, slow consumers

In the previous section we discussed that a reactive collection is equivalent to any interactive collection: they obey to the same rules and the same operators (like `map` and `filter`) can be defined. The only difference is that a reactive collection is push-based, whereas an interactive collection is pull-based. Rather than the consumer being in charge, asking for the next value, here the producer is in charge and *it* decides when to emit the next value. The consumer just has to listen and can only react to the elements emitted by the producer. This is conform the definition of a reactive collection in Section 1.1: "*Reactive programs maintain a continuous interaction with their environment, but at a speed which is determined by the environment, not by the program itself.*".

A risk that arises from allowing the producer to be in charge occurs when the consumer cannot keep up with the rate in which the producer is producing the data. This gives rise to the problem of what to do with the growing accumulation of unconsumed data.

A classic example of overproducing streams is the `zip` operator, which merges two (or more) streams by using a combiner function whenever both streams have produced an element. In this operator the problem occurs when one `Observable` always produces faster than the other. This is a problem, because `zip` cannot keep up with the rate in which the first `Observable` is producing, since the second `Observable` is much slower. The most common, but also slightly naive, implementation for this operator maintains an ever expending buffer of elements emitted by the first `Observable` to eventually combine them with the data emitted by the slower stream.

Buffering the unprocessed elements of a stream is a common answer to this problem. A fast emitting stream is draining its data into a buffer and the much slower `Observer` eventually takes the data out of the buffer. This is also what happens in Listing 1.7 with the `observeOn` operator. If the first two lines produce data in a faster pace than the last two lines can consume, the elements will be buffered inside the `observeOn`. The major problem with this buffering solution (and thus with the implementation of `zip` as described above) is that the buffer will have to use a potentially unwieldy amount of system resources.

In this section we will explore some alternatives of dealing with fast producers and slow consumers.

1.3.1 Avoiding overproduction

One way of dealing with overproducing streams is to simply avoid the problem and take proactive measures whenever this is expected to happen. In the following we will discuss several ways to reduce the amount of data by using standard operators that are defined on the `Observable` interface. For this we distinguish two types of operators: lossy and lossless operators [2].

Lossy operators

One set of operators avoids the problem in a lossy way, meaning that the data that cannot be consumed immediately will be dropped.

Throttle The `throttle(interval: Duration)` operator only propagates the first element that is received in a particular interval. All other elements are discarded. Once an interval has finished, a new interval starts immediately, in which again the first received element is propagated and all other elements are discarded.

Sample Rather than propagating the first element and discarding all others, the `sample(interval: Duration)` operator is used to discard all elements except for the last one that is received in a certain `interval`. This operator can for example be used when someone only wants to receive the newest data from a stock ticker, but only every 5 seconds, without the need to process every value that comes in between.

Debounce The operator `debounce(timespan: Duration)` will only propagate its received values after a certain `timespan` has passed without receiving any other values. When a value is received within the `timespan`, the previous value is discarded and the same process starts all over again with the newly received value. This operator is most commonly used in text fields within user interfaces, in order to avoid too much keystroke events being generated by a fast typing user. Rather than every keystroke being emitted, this operator will only yield the last keystroke after a particular `timespan`. Notice that in some versions of Rx this operator is also referred to as `throttleWithTimeout`.

With these operators many situations of potentially overproducing streams can be avoided by eliminating all the elements that do not really matter for a particular application. Note that, since these operators are depending on intervals, they have to operate on a `Scheduler`, which is supplied as an extra argument to either one of these operators.

Loss-less operators

Even though these lossy operators solve a large part of the problem, there are still as many cases left in which *all* data that is send over a stream needs to be processed. For these kinds of use cases Rx provides so-called loss-less operators. These operators can buffer or group the data and emit collections of data, such that there are at least less elements to deal with. Note that this is different from buffering as discussed at the start of this section, as loss-less operators group elements and emits them *all at once* in a collection rather than storing them as a backlog to be consumed *one by one* at a later moment in time!

Buffer The `buffer` operator comes in a couple of different overloads. First of all, the `buffer(n: Int): Observable[List[T]]` operator, which receives data and groups it into lists of `n` elements. Once a list is filled (e.g. when its size is `n`), it is emitted to downstream. Until then, the operator holds the data it receives. The `buffer` operator also comes in another form: `buffer(interval: Duration)`, which groups the data it receives within a certain interval into a single list. Just as with the lossy operators, note that this operator depends on an interval and therefore requires a `Scheduler`. Finally there is `buffer[B](boundary: Observable[B])`, which groups the data it receives between two emissions of `boundary` into a single list.

Window The drawback of a buffer is that it only propagates its received values once the buffer is filled, the boundary `Observable` fires or the interval is over. All the time in between, no elements will be received by the observer or downstream operators. This already becomes clear from the return type: `Observable[List[T]]`. In order to accomplish the same behavior but with an `Observable` rather than a `List`, the `window` operator is included in Rx, having the same kinds of overloads as `buffer`. Instead of the list being emitted only once it is completely filled, the inner `Observable` is emitted as soon as the first element is received and is *completed* once the size, interval or boundary requirement is met.

These operators form a first line of defense against overproduction. For streams where not all elements necessarily need to be processed (for instance the keyboard events on a text field), a lossy operator can be used. For streams where all emitted data is needed, a loss-less operator is the right solution. Besides that, for special occasions lossy and loss-less backpressure operators can be combined in order to create the optimal buffering strategy. This was discussed in some further detail in a conference talk at QCon by Ben Christensen [22].

1.3.2 Callstack blocking

Another way of dealing with this problem is to block the callstack and with that 'park' the thread on which the `Observable` is running. This directly slows down the producer and gives the consumer more time to process each element of the stream. Despite the fact that this approach goes against the 'reactive' and 'non-blocking' model of Rx, it can potentially be a viable option if the overproducing `Observable` runs on a thread that can safely be blocked.

This technique is currently not used in RxJava [3], but *is* used in a particular implementation of the `zip` operator in RxMobile [17]. Once either one of the streams emits a value, it is blocked until the other `Observable` has produced a value as well. In order to avoid blocking the upstream callstack completely, it is strongly recommended to switch each `Observable` to another thread or scheduler using the `observeOn` operator. With this only the callstack is blocked upto the start of this new thread or scheduler. Once the other stream has produced a value, the blocking of the first `Observable` sequence is removed and the `zip` operator waits until either one of the sources emits a next value.

It is debatable whether or not this is a really effective approach. The elements are not buffered in the `zip` operator now, but are still buffered in the `observeOn`.

1.3.3 Reactive Streams

Reactive Streams is an initiative [8] of a number of companies such as Netflix, Pivotal and Lightbend with the mission to *provide a standard for asynchronous stream processing with non-blocking backpressure.* This collaboration resulted in an alternative API [9] for stream processing that is claimed to be capable of handling overproducing streams using backpressure.

The Reactive Streams API (see Listing 1.9) looks fairly similar at first glance to the original API that was developed by Microsoft, but has some particular differences that have significant influence on the handling of backpressure. The `Observable`, renamed `Publisher`, looks the same: it still is parameterized over `T` and still has a `subscribe` method. The `Observer`, which was passed to the `subscribe` method, is replaced with a `Subscriber`, even though it is almost the same[3]. It only adds an ex-

[3]The `onComplete` method in Listing 1.9 does not contain a typing error compared to Listing 1.1. This is actually how the API is specified!

tra method `onSubscribe`, which is called right after the `Subscriber` is subscribed to the `Publisher`. This `onSubscribe` method requires an argument of type `Subscription`. This last interface contains two methods: `cancel`, which is similar to the `unsubscribe` method in Listing 1.1 and a new method `request(n: Long)`.

This last method reveals the whole idea behind the Reactive Streams API, namely to let the `Subscriber` request a certain amount of elements from the `Publisher`. This way the `Subscriber` is in charge of how many elements it will receive eventually and the `Publisher` just has to send at most this amount of elements by calling the `onNext` method.

Listing 1.9: Publisher, Subscriber and Subscription

```
trait Publisher[T] {
    def subscribe(s: Subscriber[T]): Unit
}

trait Subscriber[T] {
    def onNext(t: T): Unit
    def onError(e: Throwable): Unit
    def onComplete(): Unit
    def onSubscribe(s: Subscription): Unit
}

trait Subscription {
    def cancel(): void
    def request(n: Long): Unit
}
```

1.3.4 Reactive pull

The ideas that sprout from the Reactive Streams initiative have been incorporated in the RxJava library. They kept the original naming conventions of `Observable` and `Observer` but added a couple of new methods to the latter: `request(n: Int)`, which signals to the `Observable` that it will be able to handle `n` new elements and `onStart()`, which performs the initial request from the `Observer` to the `Observable`. After the initial request is done, the `Observable` sends at most `n` elements to the `Observer`, which receives them in the `onNext` method. This is now the place where the `Observer` can call `request` again to receive more data.

We recognize these two methods from the Reactive Streams API, where `onStart()` was called `onSubscribe` and `request(n: Long)` was inside the `Subscription` interface. Making these slight modifications does not change the intention of the Reactive Streams API: it introduces a feature called *reactive pull* in the `Observer` to manage the number of elements that are emitted by the upstream `Observable`.

Earlier versions of RxJava, who did not have this feature only had the ability to communicate upstream by calling the `unsubscribe` method. Recall that when an `Observer` unsubscribes, the `Observable` is basically signaled to stop emitting any data to that particular `Observer`. Besides unsubscribing there was no other way in the standard Rx model to communicate upstream.

This feature from Reactive Streams allows the `Observer` to have some more control by *pulling* from the data source (`Observable`) at its own pace. RxJava is set up in such a way that processing data under normal circumstances is still push based. Only when the `Observer` can't handle the speed in which data is sent, it will switch to this pull based model. Wrapping it up in this way, the problem of overproduction is not prevented or gone away but is rather moved up the chain of operators to a point where it can be handled better [6].

With this it limits the number of elements that are in a buffer within certain operators. Using this new feature, the earlier mentioned `zip` operator is implemented by using a small buffer for each `Observable`. It only requests items from one of these sources when there is room for more elements in its buffer. Once all buffers contain at least one element, the operator can remove an element from each buffer, zip these together and push them downstream. After that there is room for at least one extra element in all buffers, hence new requests are sent to the upstream.

Notice that the RxJava wiki [6] points out that this method only works when *all* streams that are zipped together respond correctly to the `request()` method. This is *not* a requirement for the normal `Observable`, but it is required for instances of `Observable` that are used in operators like `zip` that depend on reactive pull. RxJava therefore provides operators such as `onBackpressureBuffer` and `onBackpressureDrop` that respectively buffer and drop data that cannot be consumed immediately by the downstream `Observer`.

Chapter 2

Controlling overproduction

In the previous chapter we established the definition of a *reactive program* in terms of its behavior, how it is implemented in Rx, what the danger is of having an overproducing `Observable` and which solutions already exist to solve this problem. In this chapter we will discuss these solutions in further detail, consider their advantages and disadvantages and argue which solutions work for the various kinds of streams.

2.1 Hot and cold streams

Applying the definition of reactiveness by Berry [20] to the Rx `Observable`, we can conclude that every `Observable` sequence starts with a source that emits values at its own pace. No matter which function is used for this (`apply`, `range`, `timer`, `interval`, etc.), ultimately they all are the result of the `Observable.create` function. This function lifts an arbitrary source into the `Observable` interface and treats its values like streaming data. The behavior that is exposed by the resulting stream can, however, differ from source to source. We already introduced a behavioral distinction in Section 1.2.5, where we separated hot and cold streams. In this section we will further analyze this distinction and connect it with the definition of reactiveness.

Sources like clocks, stock tickers, mouse moves or key presses start emitting regardless of any `Observer` being subscribed to the stream. When no one listens, the data is simply discarded; when multiple `Observer` instances are subscribed, every one of them receives the same data at (approximately) the same time. In case an `Observer` subscribes at a later time, it will not receive all previously emitted data, but will only share in the data that is send after it is subscribed. This kind of source is considered to be *hot*. It is strictly reactive, meaning that it can only emit data at a speed which is determined by the source and has no way to be slowed down by any `Observer` that cannot cope with the amount of data sent.

On the other hand there are streams that originate from sources that are actually interactive. These include the results of database queries and `IEnumerable` sequences. Often the reason for them being wrapped into an `Observable` is because of a potential delay that needs to be awaited before the result is returned without blocking the program flow or call stack, or simply because the context of the program requires an `Observable` rather than an `IEnumerable`. Regarding the former, it may (for example in case of lazy evaluation) take some time before the `IEnumerator` has produced its next element. The `Observable` will however not start emitting its data immediately, like the hot variant, but will wait until at least one `Observer` is subscribed. Only then it will start producing its values. In case a second `Observer` subscribes, the stream will effectively duplicate itself and start all over again with emitting the first values. Unless specified by operator sequences, this means that the lazy `IEnumerator` in the source will have to produce its values a second time as well. In the end, both the first and second `Observer` have received the same set of data, even though the second subscribed much later than the first. A stream with this kind of behavior is referred to as a *cold* `Observable`.

We propose a further behavioral distinction in the class of cold streams: the *cold asynchronous `Observable`* and the *cold synchronous `Observable`*. The former is bound by a certain notion of time. This type wraps a source that is interactive, but can take some time before it produces its next element. Examples of this can be a network response, the result of a database query or an `IEnumerable` sequence where each element is computed lazily and therefore takes some amount of time to be returned. Often an `Observable` like this is executed on a different thread using `observeOn` or `subscribeOn`, in order to avoid blocking the programs main thread.

The *cold synchronous `Observable`*, on the other hand, is *not* bound by any notion of time. This kind of stream mainly wraps interactive sources for which the values have already been computed, such as a list, and will fire it's values as fast as the downstream will allow it.

2.2 Solutions for overproducing sources

In the previous chapter we gave an overview of a number of solutions for overproduction, ranging from putting data in an overflow buffer to gaining more control over the `Observable`. Some of these solutions work perfectly under certain circumstances, others do not. In this section we will analyze the solutions that were described in Section 1.3 in further detail and reflect on them in the light of the distinction between hot and cold streams.

2.2.1 Avoiding overproduction

As described in Section 1.3.1, *avoiding* is a first line of defense for overproduction. This is done by introducing several operators that either propagate only a portion of the data and drop the rest or buffer the data and propagating these buffers downstream for further processing. All *lossy* operators, as well as the `buffer` with interval, require a `Scheduler` for them to run their interval timers on, hence the output stream of these operators runs on a different thread.

For a hot `Observable` this kind of defense mechanism is perfect, as the speed at which the source is producing data is unknown. This also holds for the cold asynchronous `Observable`, as the production of elements is also bound to a notion of time. For other cold streams (like `Observable(1, 2, 3, 4)`) it does not make sense to add any overproduction-avoiding operators, as this kind of stream is sequential and will in fact emit at a rate which is determined by the downstream (this is discussed in further detail in the next section).

We are aware of the fact that these operators also have use cases other than avoiding overproduction, such as edge detection or calculating the derivative of a stream of numbers. These use cases are, however, not relevant for this thesis.

2.2.2 Callstack blocking

Subscribing to an `Observable` is basically nothing more than supplying an `Observer` to the function within `Observable.create` and *sequentially* executing this function using this provided `Observer`. Once an element is emitted by the source, all operations in the `Observable` sequence are executed on that element before a second element is emitted. In the sequence of operators in Listing 1.6, first all operations on 1 are performed, before 2 is emitted by the `Observable` (and then discarded by `filter`). This is also the case in Listing 1.7 up to line 3, after which the elements are scheduled on a different thread and hence are further processed in parallel with emitting new elements from the source.

The order of operations is designed in such a way that it allows for lazy evaluation. If a cold `Observable` contains 5 elements and the operator sequence contains the operator `take(2)` (see Listing 2.1), only the first 2 elements from this `Observable` will be evaluated by the upper half of the operator sequence, rather than all 5. After the second element has passed `take`, it sends an `onCompleted()` downstream, right after the second `onNext`, causing the stream to end without evaluating the other 3 elements.

Listing 2.1: Lazy evaluation

```
Observable(1, 2, 3, 4, 5)
    // some operators
    .take(2)
    // some more operators
    .subscribe(i ⇒ print(i + " ")) // only prints the first and second element
```

Following this order of operations, we can conclude that basically every operator in the operator sequence is in a sense blocking the callstack during its computation and thereby preventing the source from emitting a next element until the previous element has gone through the whole operator sequence. This is the reason why we concluded in the previous section that avoiding overproduction on a cold synchronous stream does not make sense: the rate of emission is already determined by the speed in which all operators together are executed.

This is also true for all kind of streams, whether hot or cold, whether or not it has latency and no matter what kind of source emits the data. This may seem quite surprising, especially for the hot `Observable`, since we always claimed that this type of stream is only controlled by its outside environment. However, an example of this is shown in Listing 2.2. Here a cold `Observable` is subscribed to a `Subject` (line 12), making it hot according to Section 1.2.6. At several points in the operator sequence the time passed since `start` is measured and printed. At line 10 the callstack is blocked for 1 second by pausing the thread on which this whole process is running. The console output of executing Listing 2.2 is provided in Listing 2.3[1].

[1]Due to the inner workings of the JVM, the times shown here may vary by a couple of milliseconds in every execution.

Listing 2.2: Applying callstack blocking on a hot `Observable`

```
def now = System.currentTimeMillis()
val start = now
def timePassed = now - start

val timer = Observable(1, 2, 3, 4)
    .tee(i => println("[" + timePassed + "] emitted - " + i))
val subject = Subject[Int]()

subject.tee(i => println("[" + timePassed + "] before - " + i))
    .tee(_ => Thread.sleep(1000))
    .subscribe(i => println("[" + timePassed + "] after - " + i))
timer.subscribe(subject)
```

Listing 2.3: Console output from Listing 2.2

```
[47] emitted - 1
[47] before - 1
[1059] after - 1
[1059] emitted - 2
[1059] before - 2
[2060] after - 2
[2060] emitted - 3
[2060] before - 3
[3062] after - 3
[3062] emitted - 4
[3062] before - 4
[4064] after - 4
```

From these measurements we see that the first value was emitted at $t = 47\ ms$. Almost immediately after that, the thread on which the whole program is running is blocked for 1 second. At time $t = 1059\ ms$ the first value is propagated to the `subscribe`. *Only then* the second item is emitted by `Observable.apply`.

From this we can conclude that even a hot `Observable` can be controlled by callstack blocking. Notice however that this creates a (potentially unbounded) buffer of unprocessed `onNext` calls within the `Observable.create`'s callstack, using an excessive amount of memory. With that we only emulate the naive solution to overproducing streams (see Section 1.3) by using callstack blocking. The same buffering behavior can also happen within the `observeOn` operator, when the other thread used callstack blocking to slow down the stream.

The case described above is one where the thread cannot be blocked safely. It can potentially blow up the program when the buffer gets too big. This is what RxJava warns against in its wiki [3] and what RxMobile warns against in the context of the particular `zip` implementation discussed in Section 1.3.2 [17]. Only certain kinds of operators can use callstack blocking safely, provided with streams that can handle this callstack blocking safely.

Another issue with a hot `Observable` is what happens when more than one `Observer` is subscribed and both do callstack blocking. This can potentially lead to even more disastrous situations, where a fast `Observer` suffers from a slower one and where deadlock situations are inevitable.

In general we can conclude that controlling the flow of data by callstack blocking is already implicitly used in cold no-latency streams, but that there is no good in using it on a hot or cold with-latency `Observable`, unless being completely certain of the stream's behavior.

2.2.3 Reactive Streams

A third way of managing the overflow of data is by using the backpressure technique that was created by the Reactive Streams initiative. The associated API, discussed in Section 1.3.3, consists of a `Publisher` that corresponds to the Rx `Observable`, a `Subscriber` that adds an `onSubscribe` method to the Rx `Observer` and a `Subscription` with a `cancel` and `request` method that replaces the Rx `Subscription`.

The way Reactive Streams handles backpressure is to let the consumer determine how many elements it wants to receive from the producer. Then the producer will send this amount or elements if and only if it is able to produce them. Note here that it therefore might send less elements if the producer is not fast enough. This is a technique that is fairly similar to the TCP protocol, which also advertises to its host how many elements it is willing to receive and buffer [40].

The most important question to ask here is whether or not the approach of Reactive Streams is a solution to overproducing sources in reactive programming, as is advertised on the website [8] and implied by their name. To answer this question, we first need to reflect on the previous paragraph: the consumer determines how many elements it is willing to receive and has to communicate that with the producer. This means that the *consumer* is in charge and the producer can at most send the amount of data that is requested by the consumer. This is completely against the description of a reactive program according to the definitions from Gérard Berry [20]. The behavior of the Reactive Streams API instead conforms to the description for interactive programs in the sense that the consumer "*interacts at its own speed with users or with other programs*"; in this case with the `Producer`.

Besides that, we need to take into consideration that 'reactive' is the dual of 'interactive'. In other words, an interactive interface needs to be *mathematically dualized* in order to become reactive. However, the Reactive Streams API can be constructed from the `IEnumerable` and `IEnumerator` interfaces without using any dualization techniques, as shown by Erik Meijer during a conference talk at Lambda Jam 2014 [34]. Instead he shows that using techniques such as coproduct, continuation passing style and currying will suffice.

Finally, we have to point out that it is not even possible (following the definitions) for the Reactive Streams API to communicate with or wrap a *reactive* source. The former has to communicate how many elements it is willing to receive, whereas the latter does not have any possibility to be interacted with! Concretely this means that Reactive Streams can for example not wrap a stream originating from mouse events or button clicks.

From these three issues we can concludes that the Reactive Streams API is not reactive at all, but is interactive instead. However, it still distinguishes itself from the classical set of interactive programming interfaces: `IEnumerable` and `IEnumerator`. Rather than calling `moveNext` and `current`, a `Subscriber` can advertise to its `Producer` that it can handle `n` more element. It is then up to the latter to send these elements either immediately or after some amount of delay to the former by calling the `onNext` method, signaling the end of the stream by calling `onCompleted` or propagating an exception by calling `onError`. Just as Rx, the API is set up in such a way that it does not block the program flow, which would be the case when using `IEnumerable` and `IEnumerator`. In that sense, Reactive Streams is definitely not the same as `IEnumerable` and `IEnumerator`, but is rather an API for *asynchronous interactive programming*; it is not just a normal pull model, it is an *asynchronous* pull model.

Following the discussion above, we can classify Reactive Streams as a overproduction solution for both synchronous and asynchronous cold sources. In these cases a `Subscriber` can pull as much data from the source as it wants, only restricted by the size of the source in case it is finite. Note that for a synchronous source the response of a request will be immediate, whereas the asynchronous source might take some arbitrary amount of time to produce its next value.

A hot source on the other hand is not as suitable for the Reactive Streams API. Following the definition of a hot stream, there is no way for the `Producer` to interact with it. It cannot request the next `n` elements from it, nor can it request to slow down or speed up. This is implied in the nature of a hot source [20]. The only way to have a hot source be wrapped in a `Producer` is to drain its the data in a buffer or queue and send the requested amount of elements from this buffer to the `Subscriber`. In general this is again that dangerous move of possibly storing unbounded amounts of data for later processing as is described in earlier sections.

2.2.4 RxJava and reactive pull

RxJava started of as a port of Rx.Net for the JVM and was until version 0.20 purely reactive. It did not have any other policies for handling backpressure than the ones discussed in Section 1.3.1. In version 0.20 backpressure support was introduced as a result of collaborations in the Reactive Streams initiative. This implementation (also referred to as *reactive pull*) did not change the original Rx interfaces but rather added extra interfaces and methods that were derived from the Reactive Streams API. Due to these changes, the reactiveness is partially lost, as the data flow is now controlled by `request(n)` calls from the `Observer`.

From version 0.20 onward, RxJava has a lot of similarities with the Reactive Streams API in terms of handling backpressure. It is therefore well equipped to handle cold sources by wrapping them in the `Observable`. However, in contrast to Reactive Streams, this API is still able to correctly cope with hot sources, even though it explicitly declares that backpressure is *not* suitable for this kind of stream [5]. The RxJava wiki states that for this to be enabled, the `Observable` needs to be initialized with `request(Long.MAX_VALUE)`, which orders the `Observable` to emit data at its own pace [6]. It also advises to use other flow control strategies in general for a hot stream, mainly the ones discussed in Section 1.3.1, as well as the `onBackpressureDrop` and `onBackpressureBuffer` operators that were mentioned in Section 1.3.4.

While introducing backpressure support to RxJava, it turned out that all operators that needed to support this feature had to be reimplemented for the `request(n)` to be used. In some cases (like `take(n)` or `skip(n)`) this can be a simple straightforward refactoring or an extra method override. However, in other cases it is much more complicated to do this. For example, since the `Observable` is a monad, it has to implement `flatMap`, which in the world of streams means mapping and merging. But how can one implement this `merge` operator in terms of requests? When `n` elements are requested, to which of the `m` upstream `Observable` sequences is this request sent? Is it sent to only *one* of these, with the possibility that this is a rather slow producer; or is the `request(n)` split into multiple smaller `request(k)`s and sent to multiple upstreams with the possibility of getting less than `n` results; or is the `request(n)` sent to all upstreams with the possibility of getting much more than `n` elements? All of a sudden the implementation is no longer straightforward and easy, as we would like it to be, but becomes much more complicated by making decisions like this. RxJava chose an approach with round-robin techniques to implement `merge`. This however caused a giant increase in code complexity [16], making the code almost unreadable and not particularly suitable for any form of maintenance.

Similar problems with timing, mouse moves and operators like `groupBy` are discussed in [34].

The underlying issue of this code complexity is the coupling of this approach with the operators. When the new strategy of backpressure needed to be introduced in the 0.20 version of the RxJava code base, it automatically followed that the operators had to be changed as well. This was mainly caused by change in direction of control, which went from '*the producer is in charge*' to '*the consumer is in charge*'.

As backpressure is mainly suitable for cold sources, not all operators are able to support backpressure, since they convert to hot streams or use another flow control mechanism like time or another stream. Detailed reasons per operator for not implementing backpressure can be found in the RxJava documentation [13].

2.2.5 Conclusion

In general we can conclude that it differs per situation what can be done to avoid overproduction. We can conclude that by definition a hot source does not allow for interaction, neither does it provide any way to request it to either speed up or slow down. Backpressure as invented by the Reactive Streams collaboration and used in RxJava does not take this kind of sources into account. The only way to make these two work together is to drain the source in a buffer and wrap the Reactive Streams `Producer` or an Rx `Observable` around this buffer. Unless preventive actions are taken, this can cause an unwieldy amount of data being stored in memory, which is exactly the problem that backpressure is trying to solve.

The best solution for managing a stream and guarding for overproduction in cases of a hot stream is either bundling the data in larger groups and processing it in these groups or discarding some of the data that cannot be processed immediately or shortly after it was received. These solutions are already present in the Rx API as the lossy and lossless operators discussed before.

Cold sources on the other hand, regardless of being synchronous or asynchronous, work well and are better suited for techniques like backpressure. With these sources the consumer can leverage their interactiveness by advertising to the producer how much data it is currently willing to accept. For these sources the Reactive Streams API works fine and is well suited. The fact that this causes the whole stream to being not reactive by definition is not important here, since the source itself is not reactive.

Chapter 3

Introduction to feedback control

Control is something we deal with on a daily basis. Whether it is the temperature in our houses, the number of people in line at the supermarket's checkout or something more critical like the number of neutrons emitted during a nuclear reaction in a reactor, we have to control it to avoid potential chaos.

In software development we too want to have control over the products we both create and use. For example, most applications have a settings menu where the user can alter the application's behavior to their liking; to ensure our code to be working correctly we use compilers, IDEs and automated testing tools that help us trace errors and bugs; companies like Google, Facebook and Twitter do A/B-testing and other kinds of user studies when introducing new features and with that control and improve the usability of their product; and last but not least, in cloud computing we want to control the number jobs in a queue and scale up or down depending on the amount of jobs in the queue, such that all jobs get executed as fast as possible with spending the least amount of resources.

A common factor in controlling a system is the notion of feedback: you change something in the environment and see how well it adjusts towards your desired outcome. If the lines at checkout get too long, most likely an extra cash register needs to be opened; if your users don't perform well on an A/B-test of your latest features, you should probably revise these features rather than bringing them into production; and when you alter the settings in an application, you check whether the new behavior is more to your desire and maybe fiddle around with them some more until it is more to your liking.

Controlling a system without the notion of feedback is possible, however this requires an exact model of the system under control. External forces that are not part of the model cannot be allowed to disturb the system. In most cases this model does unfortunately not exist and control has to incorporate the system's output to come up with it's next input. If the output is not taken into account, the system can drift off and end up in undesired situations. A nice example of this are Microsoft's user studies: the user's feedback was not taken into account when Office 2007 introduced the “ribbon” to replace the existing user interface nor when the start button disappeared in Windows 8 and suddenly every Windows user lost their 20+ years of accumulated muscle memory for commanding the Windows operating system [36].

In physics and engineering, feedback control is a commonly used technique and is also fully backed by mathematics. A series of equations and theorems such as Laplace Transforms and differential equations describe the abstract behavior of a feedback controlled system and allow to evaluate its outcome after running it for a certain time and with a certain input [26, 28]. These theorems and equations are usable in science because everything else is described in the ‘language of mathematics’ as well. We have known the differential equations to Newton's law of cooling, damped harmonic oscillators and so forth for centuries, as well as their Laplace Transforms, which are needed for the ‘feedback equations’.

This is all in contrast to the current situation in computer science. Although proven to be very effective in physics, computer science has not yet widely adopted the techniques of feedback control [28]. Instead, complex algorithms are written to control a web cache or do cloud scaling. Although it seems very naturally applicable, it is surprising to find that such a simple and effective technique is mostly ignored.

A potential reason for this lack of using feedback control lies in the fact that computer science does not yet fully understand the whole mathematical background of the datastructures and applications that are being used and created. Given this lack of understanding, we are not yet able to describe to which laws our datastructures and application obey. What would be the equivalent of Newton's laws for a web cache [28]? Only recently some work has been done on understanding the mathematics behind this [19] and we may very well see more of this in the future, which will eventually enable computer science to create a more formal mathematical model with theorems and equations equivalent to physics.

The lack of applying feedback control in computer science is also reflected in this technique commonly not being taught in university degree computer science education. Although it is one of the standard courses in physics and engineering studies, it is not at all present in the computer science curricula. And in the rare occasions it is taught, it is done in a way that a physics student would learn about it, namely as yet another maths course, even though this way of treating feedback control in computer science is not really applicable!

In this chapter we will introduce the basic concepts of feedback control with as little mathematics as possible. We will start with an overview of what a feedback system consists of and how it works. Next we will go into some detail on how to control a feedback system. Finally we will introduce a simple toy example that demonstrates the power of feedback control and translates these concepts into an imperative style program.

3.1 Basics of feedback control

In general a system is controlled by feedback when its next input value is (partially) determined by its previous output(s). The reason for taking the previous output into account while coming up with a next input is due to external forces that may impacting the system in unexpected ways. These external forces do not come regularly, are not predictable and are also not equally strong each time, hence a notion of uncertainty in the system's behavior occurs. Due to this uncertainty, a model that accurately calculates the system's next input either does not exist or would be too complex. The solution for incorporating the uncertainty from external forces is to use a feedback cycle around the system: the system's output (affected by both the system's input and any external forces) is measured and compared to a desired reference value, after which their difference is transformed into the system's next input.

Important to mention here early on is that controlling a system with a feedback cycle is not a solution to optimization problems. Tasks like "*Make the flow through the system as large as possible*" cannot be accomplished with feedback loops [28], as they compare the system's current output with a known reference value. Tasks like this require some kind of optimization strategy that determines the reference value, after which a feedback loop can be used to bring and keep the system's output in this desired state.

The input and output of a feedback controlled system are not to be confused with the actual input and output of this system. The air conditioning or heating system has an actual output of hot or cool air, whereas the feedback system's output can be any other measurable metric such as the new temperature after the actual output is applied or the temperature difference caused by the actual output. The input and output of a feedback controlled system are often referred to *control input* and *control output*.

3.1.1 Calculating the next control input

Every time the control output produces a new value, it is compared to a reference value or *setpoint* for it to calculate the *tracking error* as the deviation of the control output from the setpoint:

$$\text{tracking error} = \text{setpoint} - \text{control output} \tag{3.1}$$

This tracking error is then transformed into the next control input by the *controller*. When the tracking error is positive (thus the control output is less than the setpoint), the controller has to produce a new control input that ultimately raises the control output to the same level as the setpoint, such that the tracking error becomes zero. Of course this depends on the *directionality* of the system: for some systems the control input needs to be lowered for the system output to be raised. The controller must be able to make this distinction and therefore has to know the directionality of the system. For example, a heating system requires the controller to *raise* the control input to get a higher temperature, whereas a cooling system requires the controller to *lower* the control input to get a higher temperature.

Besides the directionality, the controller also needs to decide on the magnitude of the correction. If the magnitude is too high, the controller could overcompensate and turn a positive tracking error into a negative tracking error and vice versa, causing the system to oscillate between two states. The worst situation occurs when the negative tracking error caused by the overcompensation of the positive tracking error is larger than the positive tracking error: this results in an ever growing amplitude of the oscillation, which eventually makes the whole system unstable and in the end causes it to blow up.

On the other hand, the magnitude can be too low, causing the controller to undercompensate. This causes tracking errors to persist for a longer time than necessary and makes the system respond slow to disturbances. Although this is less dangerous than instability, this slow behavior is unsatisfactory as well.

In general we require the controller to take in a tracking error and come up with a new control input such that the tracking error will go to zero as soon as possible. For this it turns out that the controller does not need to know anything about the controlled system, but only requires information about the directionality of the system and the magnitude of the correction.

3.1.2 Architectural overview

The architecture of a feedback system is usually depicted as a set of boxes connected with arrows. This way we get a quick overview of the design without worrying about the exact implementation of the various components. The general architecture of a feedback system as discussed above is shown in Figure 3.1. Notice that here the control output is negated on the way back and is then added to the setpoint in order to calculate the tracking error. This is common practice as some more preprocessing is required before comparing the control output with the setpoint. For example, the output may contain noise which needs to be smoothened by some kind of filter. Of course preprocessing steps will also be drawn in this overview if applicable.

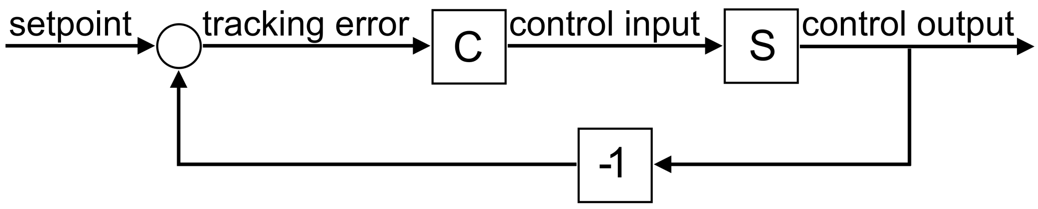


Figure 3.1: The architecture of a feedback control system

Besides these filters, a controller may consist of multiple components by itself. When using an incremental controller, only the difference in the control input is outputted. For the actual input we need to maintain a running sum of all the previous controller outputs and in that way calculate the actual control inputs. Usually these extra steps are also depicted in the architectural overview as extra boxes and arrows.

3.2 Different types of controllers

The control input in a feedback system is the value that tells the system under control to behave slightly different than it did and possibly incorporating external disturbances on the system in a better way. As discussed in the previous section, this control input value is computed in the controller part of the feedback system. Although one might expect differently, the controller itself does not turn out to be very smart or know a whole lot about the system under control. As mentioned before, a controller in fact only needs to know about directionality of the system and the magnitude of the correction for it to work just fine. There are a number of controllers that only need these two pieces of information and that are commonly used in physics, mechanics and electronics.

3.2.1 On/Off control

The most simple controller one can think of is just an *on/off switch.* Whenever the tracking error is positive, the controlled system is turned on and when the tracking error becomes negative it is turned off again[1]. For simple systems this kind of control will suffice, although it will not be a very effective approach.

An application of this kind of control might be a air conditioning system that turns on when the temperature exceeds a preset level. In this example the control output is the current temperature in the room, the setpoint is the preset temperature, the control input is a boolean value which determines whether the system should be on of off and the directionality of the system is negated. Imagine the temperature initially being much too high, causing the air conditioning to turn on right away. After a certain amount of time the temperature reaches the desired setpoint (the tracking error becomes zero), hence the system will shut down. Shortly after the air conditioning system is shut down, the temperature starts increasing again. This immediately causes a deviation from the setpoint, forcing the air conditioning to turn on again. Soon enough the temperature is low enough again for the system to be turned off, after which the cycle starts all over again. Of course this kind of behavior is very annoying to everyone working in this room, as the air conditioning continuously turns off and on again. Besides that, this behavior costs an unnecessary amount of energy for turning the system on and off, which is not really desirable either.

Obviously this behavior is caused by the controller that dictates to turn off the system whenever the setpoint is met. Due to external disturbances such as whether conditions the feedback system is off track soon again, which causes the controller to decide to turn the controlled system back on.

Small improvements that are often used in these kinds of systems are introducing a dead zone or Schmitt trigger, which causes the system to continue with the same corrective action until a certain threshold or a certain amount of time is exceeded. This prevents the controller from overreacting on sudden and short spikes in the control output. In the case of the air conditioning an addition such as this will cause the controller to not immediately send a 'turn off signal' when the setpoint is zero and will

[1]Of course this is dependent upon the directionality of the system under control.

not activate the system with the slightest deviation, but instead waits a little longer until a threshold is exceeded before activating or deactivating the system again.

3.2.2 Proportional control

Another improvement on the on/off controller is to change the output. Rather than sending a boolean value to the controlled system, a real controller should send a floating point value. The *proportional controller* does this by taking the magnitude of the error into account when calculating the magnitude of the corrective action. If the tracking error is small, the control input should be small and if a large tracking error occurs, the control input should be proportionally large. To achieve this, the proportional controller uses the following formula:

$$u_p(t) = k_p \cdot e(t) \quad \text{where } k_p > 0 \text{ constant} \tag{3.2}$$

Here $u_p(t)$ and $e(t)$ are the control input and the tracking error at time t respectively and k_p is the proportional controller gain, which is a positive constant. The control input is thus calculated by multiplying the *current* tracking error with the controller gain, hence the control input is proportional to the tracking error.

In practice this kind of controller in a feedback system has the effect of applying small changes to the control input when the tracking error is small and larger changes for a bigger tracking error. Although this may work just fine for some systems, other system show the same kind of problem as with the on/off controller. For this we need to study the behavior of a proportional controller when the tracking error approaches zero: due to Equation (3.2), the control input becomes proportionally smaller and eventually becomes zero at the moment the tracking error becomes zero. At this point in time the feedback system has reached its desired state and (needless to say) has to keep the control output there as close as possible. However, the tracking error is zero and hence the control input is zero. For systems like the air conditioning this is equivalent to turning itself off. As with the on/off controller, turning off the air conditioning causes the temperature to rise again due to the external disturbances on the system. With that the tracking error becomes non-zero again and gives rise to the controller to turn the system back on, to only get the tracking error back to zero and turning off the air conditioning.

Although the proportional controller is a slight improvement in regard to the on/off controller in the way it translates the tracking error into the next control input as well as in the more fine-tuned way of representing this control input, it still shows the problem of not working correctly in the case of approaching the setpoint value. This problem will however be fixed in a later section.

3.2.3 Integral control

A third way of coming up with the next control input that is quite common in feedback control is to not only look at the current tracking error, but also at all the previous tracking errors. *Integral control* does this by using the running sum of both the current and all previous tracking errors. As is well known from mathematics, in a continuous stream a sum becomes an integral (hence the name), resulting in Equation (3.3).

$$u_i(t) = k_i \cdot \int_0^t e(\tau)\mathrm{d}\tau \quad \text{where } k_i > 0 \text{ constant} \tag{3.3}$$

Here $u_i(t)$ and $e(\tau)$ are the control input and the tracking error at times t and τ respectively and k_i is the integral controller gain, which is a positive constant.

Although Equation (3.3) is useful for continuous systems that occur in physics and engineering, it does not apply to computer science as in that field everything is discrete. Therefore Equation (3.4) will be used instead, which is mathematically the discrete counterpart of the integral.

$$u_i(t) = k_i \cdot \sum_0^t e(\tau) \quad \text{where } k_i > 0 \text{ constant} \tag{3.4}$$

All by itself the integral controller is not very useful, as the results (tracking errors) of all previous iterations of the control loop are still incorporated in the same proportion as the result from the current iteration. In his book, Janert [28] briefly shows a very simple toy example for this controller working all by itself, but here no external forces are applied on the system and the model is fully known. In most situations the integral controller is however not practically usable by its own. However, as we will see in Section 3.2.5, combining it with other types of controllers creates a very powerful tool that will turn out very effective.

3.2.4 Derivative control

The proportional controller and the integral controller in the previous sections respectively took the present and the past values for the tracking error into account when coming up with the next control input. The basis for the latter is founded in the mathematical concepts of integrating the tracking error. It should therefore not come as a surprise that another kind of controller can be created that tries to predict the future values for the tracking error and takes these predictions into account when calculating the next control input. This is done by calculating the derivative of the tracking error signal and using that as a way to predict the next tracking errors. From mathematics it is well-known that if the derivative of a signal is positive, the value of the signal is currently growing (and vice versa). This can be applied to the tracking error signal, from which actions can be taken to bring the signal closer to the desired value.

Mathematically we can express the derivative controller by the following equation:

$$u_d(t) = k_d \cdot \frac{\mathrm{d}e(t)}{\mathrm{d}t} \quad \text{where } k_i > 0 \text{ constant} \tag{3.5}$$

Here $u_d(t)$ and $e(t)$ are the control input and the tracking error at time t respectively and k_d is the derivative controller gain, which is a positive constant.

Although predicting the future sounds promising, Janert [28] points to a number of problems with the derivative controller. First of all, sudden changes in the setpoint will momentarily cause very large spikes in the output of the derivative controller (a.k.a. the control input), which are sent to the system under control. This effect is also known as *derivative kick*. On the other hand, many systems produce a control output signal that is full of high-frequency noise. Taking the derivative of such a signal will cause an enhancement of the effect of the noise. For this reason it is often a necessity to smooth the signal, which in itself brings both extra complexity and the risk of defeating the purpose of having derivative control with it, as over-smoothing the signal will eliminate exactly those variations that the derivative controller was supposed to pick up.

3.2.5 Combining controllers

So far we have discussed four types of controllers, of which one calculates a boolean value and the others calculate a floating point number. All of these controllers by them selves turned out to have various problems that caused them to be unusable individually in a feedback control system. One of the most prominent problems is caused by the proportional controller not working correctly when approaching the setpoint. This is due to the fact that the control input approaches zero as the tracking error approaches zero.

This problem is often solved by combining the proportional controller with the integral controller. Here the tracking error is sent to both the proportional and integral controller after which their outputs are combined by addition (Equation (3.6)). This newly composed controller is often referred to as a *PI controller*. The integral term of this controller takes care of the problem by providing a nonzero contact offset, such that when the proportional term approaches zero, the integral term will still be there and cause the PI controller to produce a nonzero output.

$$u_{PI}(t) = k_p \cdot e(t) + k_i \cdot \sum_{0}^{t} e(\tau) \tag{3.6}$$

When the air conditioning system is controlled by a feedback loop with a PI controller, it will try to bring the temperature to the desired value and with that make the tracking error as small as possible. When the desired temperature is reached the system will *not* turn off, but rather keeps running, presumably on a low enough setting such that the temperature stays the same. When an external disturbance causes the temperature to rise, the tracking error becomes nonzero again, which makes the proportional controller more active again and makes both components of the PI controller contribute to bringing the temperature back to the desired level.

Another combination of controllers is referred to as a PID controller. This combines the powers of the proportional, integral and derivative controller, such that this controller can take present, past and future into account when coming up with the next value for the control input. A common representation of the PID controller is shown in Figure 3.2. Here it is clearly visible that all three components receive the same tracking error signal, after which all three controllers produce their own output, which eventually get combined by addition (Equation (3.7)) into the controller output and control input.

$$u_{PID}(t) = k_p \cdot e(t) + k_i \cdot \sum_{0}^{t} e(\tau) + k_d \cdot \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{3.7}$$

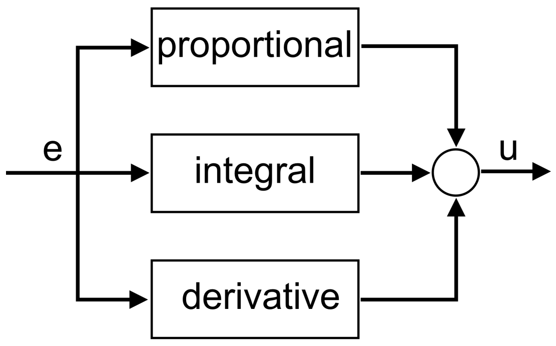


Figure 3.2: Combining proportional, integral and derivative control

The PID controller is one of the most widely used controllers in feedback control, as it has proven to be a very effective and successful controller. It is an easy concept that does not require a lot of computational power and does not require any knowledge of the system it is controlling other than the magnitude of the correction and the directionality of the system.

A drawback to these composed controllers concerns the controller gains that are involved in each part of the controller. With the PID controller we have to come up with three separate values that each determine the level of involvement of the different components in the whole controller. These three constants together determine how well the whole feedback system is working and a careful choice for these values is therefore crucially important! The topic of controller tuning will however be considered outside the scope of this thesis. Regarding this topic we refer to both Hellerstein et al. and Janert [26, 28] for extended and in-dept discussions.

Although very useful in many cases, the PID controller should not be seen as a holy grail with which every feedback control problem can be solved. Janert [28] demonstrates a case study where both the PI and PID controller turn out to be fairly ineffective. This turns out to be the case especially when a feedback system is working on integer control outputs rather than floating point numbers. In such cases a special kind of controller is necessary to get the most out of a feedback system. We will discuss this in more detail in Section 5.1.

3.3 Extended example

To get a better feel for how a feedback control system works in practice, we will discuss a simple but interesting application of feedback control that uses the theory covered in the previous sections. In this section we show an imperative reference implementation. In the next chapter we will continue to use and refactor this application as we come up with an API for constructing and executing feedback systems. The application at hand is a port from the original implementation by Nikita Leshenko in Javascript, CSS and HTML [29]. We will however use Scala as our programming language and JavaFx for drawing the graphics on the screen. To get a feel of what this application is doing we strongly recommend to have a look at the online version[2] first before proceeding with this section!

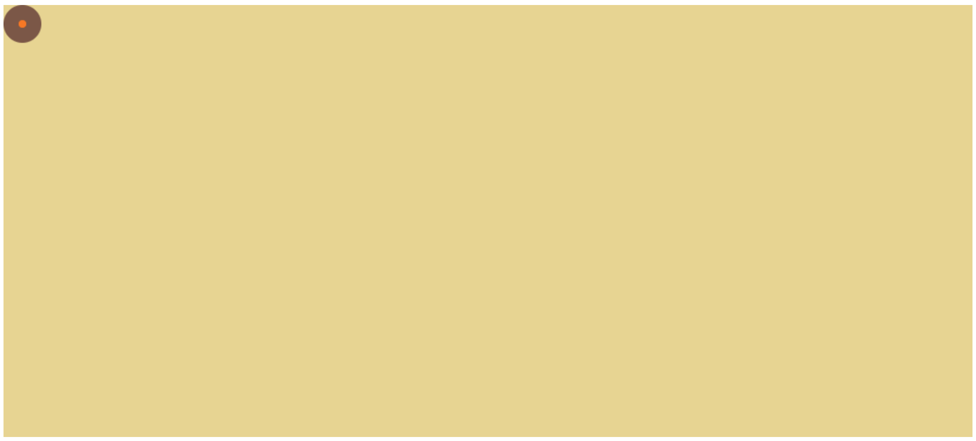
The application consists of a flat surface on which a ball can move around. The goal is to move the ball from its initial position to the position on the surface that the user clicks on with the mouse. Figure 3.3a shows the ball in its initial state when the application starts. When the user clicks on the screen, the ball starts moving to that position as shown by Figure 3.3c. Besides the background and the ball, also the desired position, a dashed line between the ball and the desired position, the acceleration in both the x and y directions and a trail of previous positions are drawn (which are fading away over time). After some time the ball is at its desired position and waits there for a new position to navigate to, as is shown in Figure 3.3d. Notice that the desired position can also be updated while the ball is still moving, in which case it moves to the new and discards the old destination.

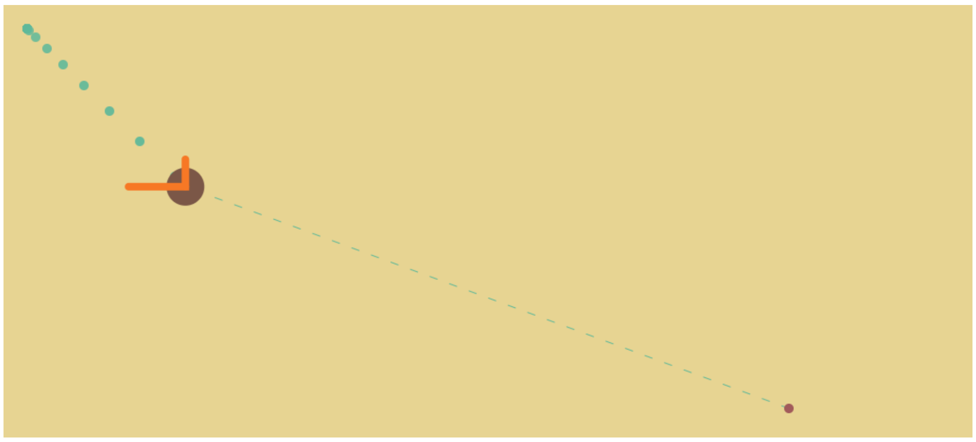
In order to allow the ball to move in a natural way, we remind the reader of some basic equations from physics that describe the one dimensional motion of an object as a function of time (Equation (3.8)). Here x_t, v_t and a_t are the ball's position, velocity and acceleration at time t respectively. For two-dimensional motion we can combine two sets of these equations for both dimensions.

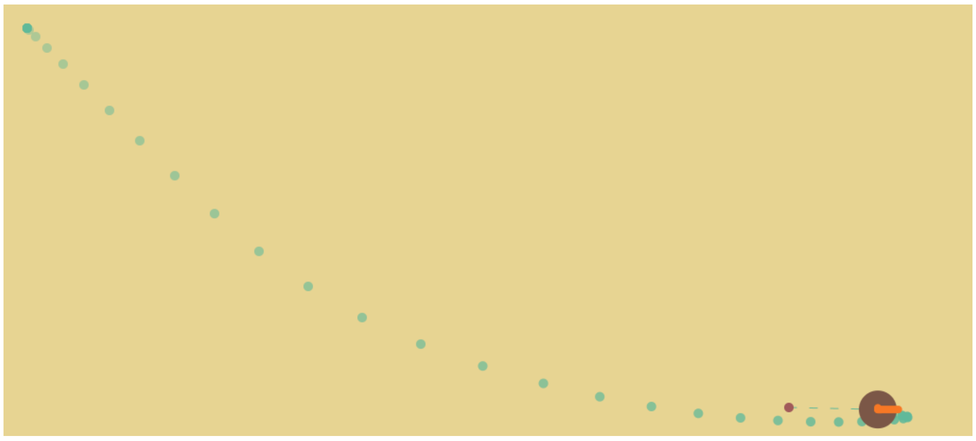
$$x_t = x_{t-1} + v_t \cdot \Delta t \tag{3.8a}$$

$$v_t = v_{t-1} + a_t \cdot \Delta t \tag{3.8b}$$

[2] http://nikital.github.io/pid/


(a) Initial


(b) Moving to desired position


(c) Moving to desired position, overshooting a bit

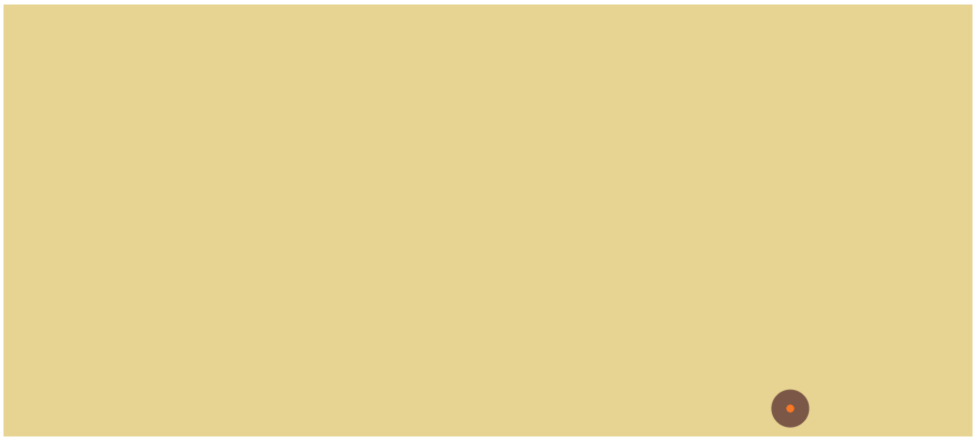

(d) Reached desired position

Figure 3.3: Ball movement

The implementation of these required pieces of physics can be found in Listing 3.1. We first define a `Position`, `Velocity` and `Acceleration` as tuples of `Double` as well as some mathematical operations on the tuple type. The `Ball` class describes the state of the ball having a position, velocity and acceleration. A second constructor (`apply`) defines the initial position with no acceleration or velocity. The method `accelerate` on this class takes a new acceleration and calculates a new state for the ball according to Equation (3.8). Notice that we discard the Δt term, since this will always be equal to 1. To keep track of the ball's previous positions we define `History` to be a queue of `Position`s.

Listing 3.1: Ball motion physics

```scala
type Position = (Double, Double)
type Velocity = (Double, Double)
type Acceleration = (Double, Double)
type History = mutable.Queue[Position]

implicit class Tuple2Math[X: Numeric, Y: Numeric](val src: (X, Y)) {
  import Numeric.Implicits._
  def +(other: (X, Y)) = (src._1 + other._1, src._2 + other._2)
  def -(other: (X, Y)) = (src._1 - other._1, src._2 - other._2)
  def *(scalar: X)(implicit ev: Y =:= X) = (scalar * src._1, scalar * src._2)
  def map[Z](f: X => Z)(implicit ev: Y =:= X): (Z, Z) = (f(src._1), f(src._2))
}

case class Ball(acc: Acceleration, vel: Velocity, pos: Position) {
  def accelerate(newAcc: Acceleration) = Ball(newAcc, vel + newAcc, pos + vel +
      newAcc)
}
object Ball {
  def apply(radius: Double) = Ball((0.0, 0.0), (0.0, 0.0), (radius, radius))
}
```

The next step in creating this application is to design the feedback system itself (Figure 3.4). The *system under control* here is of course the ball, from which at any point acceleration, velocity and position can be measured. Given that our *setpoint* is equivalent to the position of where the ball should end up eventually, it makes most sense to define the *control output* as the current position of the ball. From this it follows that the *tracking error* represents the distance to be traveled before reaching the desired position. Depending on the distance, a controller can then decide how much it wants to speed up the ball by providing a new acceleration as the system's *control input*. To get the ball's next position, this acceleration has to be integrated twice to get the ball's position after Δt time.

To transform a distance into an acceleration, we can use the power of the PID controller. As discussed before, this controller is most commonly used because of its effectiveness and simplicity. For this use case it makes absolute sense to use the PID controller as well. The *proportional controller* will look at the current distance to be traveled and comes up with some amount of acceleration. However, this acceleration will approach zero as the ball approaches its destination, causing it to keep its same velocity rather than slowing down. To prevent this, we need a strong *derivative controller* that can counteract this velocity and slow down the ball as the distance is becoming smaller. For the purpose of looking back at previous distances we also add an *integral controller* into the mix.

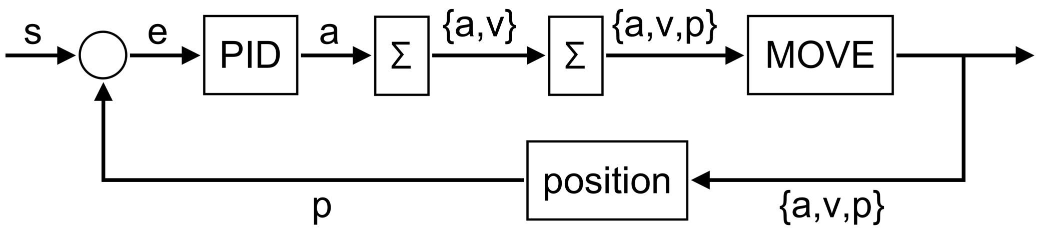


Figure 3.4: Architecture of the ball movement control system

Both the PID controller and the feedback system are implemented in Listing 3.2. After some initialization we define an iteration of the controller following Equations (3.2), (3.4) and (3.5) in the `pid` function on line 8. Again notice that this operates on the tuple types and uses the operators defined in Listing 3.1. This function returns a new `Acceleration` which corresponds to the control input as described above.

The PID controller is then used in the construction of the feedback cycle (line 19). It is first scaled down in such a way that it has an absolute maximum acceleration 0.2, after which it is fed into the controlled system. This calls the `accelerate` function on the `Ball` (Listing 3.1 line 15), which calculates the ball's new velocity and position. The new state is then stored in the `ball` variable, which represents the latest control output. The rest of this function deals with managing the history and redrawing the application, which are considered not relevant for the implementation of the feedback system itself. The omitted code can be found in Appendix A.

Now that the feedback cycle is implemented, we can plug this in a JavaFx application that contains the canvas on which the application is drawn, as well as assigning the setpoint value and looping through the feedback cycle every 16 milliseconds. The code for this can be found in Appendix A as well.

Notice that with this feedback system we lack the notion of an external disturbance on the system under control. In this case there is none since the system is just the two *sigma*s and the *DRAW*. One could argue that the changing setpoint is kind of a disturbance, although this is by definition not the case. An example of an external disturbance on this particular system could be the terrain not being flat but rather containing hills and valleys. This would influence the ball's movement because of gravity, which would add a force in a third direction. For the sake of simplicity of this example we decided to not add this feature and stick with the flat terrain. We leave implementing this feature as an exercise to the reader.

Listing 3.2: Ball drawing

```scala
var ball = Ball(ballRadius)
var setpoint = ball.position
var prevError, integral = (0.0, 0.0)

// initializing history
// ...

def pid: Acceleration = {
  val (kp, ki, kd) = (3.0, 0.0001, 80.0)
  val error = setpoint - ball.position
  val derivative = error - prevError

  integral = integral + error
  prevError = error

  (error * kp + integral * ki + derivative * kd) * 0.001
}

def update(implicit gc: GraphicsContext): Unit = {
  val acceleration = pid.map(a => math.max(math.min(a, 0.2), -0.2))
  ball = ball.accelerate(acceleration)

  // managing the history
  // ...

  // drawing all the elements
  // ...
}
```

Chapter 4

An API for feedback control

While studying the principles of feedback control, we discovered that there are hardly any publicly available libraries or APIs that abstract over the notions of feedback control, allowing us to create and execute feedback systems, both in simulation and in practice. Although surprising at first, this is completely in accordance with the earlier observation that feedback control is not (yet) a commonly used technique in computer science[1]. Surely, we can write the code ourselves as shown in Section 3.3, but that isn't really reusable, creates the danger of copy-paste behavior and is more prone to bugs than a dedicated API.

In this chapter we present our own API for creating and executing feedback systems that can potentially be used in production software. We start off with some related work and describe why there is a need to create a better API. Then we continue with a derivation of the general interface and present the various combinatorial operators that are defined on this interface. Finally we apply the API to the previous chapter's example of *ball movement control*.

The API described in this chapter is created using the (currently closed-source) RxMobile library [17]. We have created a similar API using the (open-source) RxScala library, called *feedback4s* [25].

4.1 Related work

One of the few libraries we found is described by Philip Janert in his book "*Feedback Control for Computer Systems*" [28]. Janert presents a small framework for simulating feedback loops with the purpose of being a teaching tool that makes it simple and transparent to demonstrate the algorithms presented in several case studies and to encourage experimentation. He explicitly states that little emphasis was placed on elegant implementations or time efficiency and that this framework is not meant to be used in production software. What distinguishes his framework from other simulation frameworks for control systems (for example MatLab [7]), however, is the way that the components that make up a feedback loop are implemented. As discussed before, most feedback systems in physics and engineering are described in terms of complex mathematics, using transfer functions and Laplace transformations, that operate in the frequency domain. Janert's API, however, allows algorithms to be implemented in the time domain, which is a more natural representation to software developers and people that are not familiar to the mathematics based approach.

One of the central aspects of Janert's framework is the `Component` class which contains two methods with the following signatures: `def work(u: Double): Double`, which encapsulates the dynamic function of a component, called once every feedback cycle and `def monitoring: String`, which is just a convenience function and allows for a uniform approach to logging. All components (controlled systems, controllers and more advanced building blocks like actuators and filters) are just subclasses of `Component` that implement these two methods. The feedback system is then simulated using a loop that iterates over time stamps and which calls the `work` method on the next `Component` with the result of calling the `work` method of the previous `Component`. Finally at the end of each loop cycle the result is printed to the console using the `monitor` method of the `Component` that represents the system under control.

As pointed out by Janert, this framework does well for simulation but is not really useful for production software. One concern with this framework is that it performs a feedback cycle with regular intervals between each other. However, one can easily imagine a controlled system which produces a control output very irregularly. In that case the feedback cycle has to respond to the emission of a new data point rather than asking the controlled system for its next input. Another concern related to this is the ability to handle concurrency appropriately. A component in the feedback loop might perform some kind of timing related work that requires running on a different thread or thread pool. Finally we should note that the

[1]Even though we were not able to find existing APIs for this purpose, we would not be surprised if companies turned out to have libraries like this in private.

subclasses of `Component` have to use mutable state if they want to store any data. Think of the *Integral Controller*, which has to store its current sum. Mutable state is something that computer science has come to terms with as not being so practical in some cases as we once thought it would be. Especially when introducing concurrency, mutable state is something you want to avoid at all cost!

4.2 Towards a feedback API

If we want to develop a library or API for feedback control we should keep a couple of things in mind. First of all the API should be production worthy: it should be able to drive production software/hardware and it should be easy for the developers of those systems to put together a feedback loop, without having to understand much about the mathematics that is going on in theoretical control theory. Also the concerns mentioned in the previous section should be kept in mind: the API should not rely on any form of mutable state if this is not desired by the developer, it should be easily able to support concurrency and asynchronicity and should not rely on any regularity in feedback cycles.

The idea by Janert to model a feedback system in the same way as it is drawn (see for example Figure 3.4) is something that looks very appealing to use as a foundation of our API. This means that we create a feedback system by composing smaller entities or by chaining transformations. In the same way for example Scala's collection API is written, using (monadic) operators to transform and manipulate a collection. Likewise we will use higher order functions to compose feedback systems.

Some of the basic composition operations that are required for this to work are *sequential composition*, which passes the output of the first component as the input of the second component, *parallel composition*, which passes its input to multiple components and combines the output of each component into a single value using a combinator function, *merging the output of two components* into a single value and *feeding back the output of a component to its input* to create a circuit or feedback loop. Other composition operators that one can imagine are operators like the ones found in monadic APIs in Scala or Java 8 such as collections, `Future`, `Try`, `Option` or the Rx `Observable`. Examples of these are `map`, `filter`, `take`, `drop` and `scan`.

In this section we will derive and implement an API for feedback control that uses the `Observable` as its foundational type and Haskell's `Arrow` type class as the way to do composition over the components of the feedback system. The choice for `Arrow` here seems very natural as it already defines operators for *sequential* and *parallel* composition. It also provides the *feedback loop* composition in a related type class.

4.2.1 Derivation

To get to a suitable API we must first of all observe that a component as described by Janert is isomorphic to a Mealy Machine [30]. This is a special variant of the finite-state machine whose output values are determined by both its current input and its current state. It can mathematically be described as a 6-tuple $(\Sigma, \Gamma, S, s_0, \delta, \omega)$ [21] where

- Σ denotes the input alphabet
- Γ denotes the output alphabet
- S denotes the finite nonempty set of states
- s_0 denotes the start (or initial) state; $s_0 \in S$
- δ denotes the state transition function; $\delta : S \times \Sigma \to S$
- ω denotes the output function; $S \times \Sigma \to \Gamma$

We can define an intuitive mapping from this mathematical description of a Mealy Machine to Janert's component to show their relation. Observe that Σ and Γ are the component's input and output types (which are of type `Double`). The mutable state that is implicitly present in the component can be denoted by the set of internal states S in the Mealy Machine, with s_0. For a component like the *On/Off Controller* this mutable state is a `Boolean` and therefore $S = True, False$, while the *Integral Controller* holds the current sum of all past elements and thus maps to a set S of all *Double* values. More complex states can be created by applying categorical principles of product and co-product. Transition function δ and output function ω are generalized versions of the `work` method on Janert's component. These transform the input, combined with the component's current state, into an output and a new state. For the *Integral Controller* this transformation consists of adding the input to the current sum, saving this new cumulative as the component's new state as well as returning this new state.

Note that the two functions δ and ω can be coalesced into a single function $\lambda : S \times \Sigma \rightarrow S \times \Gamma$. This function λ can also be written as a type definition in Scala:

```scala
type Component[S, I, O] = (S, I) => (S, O)
```

Notice that we use `I` and `O` to denote the input alphabet Σ and output alphabet Γ respectively. Equivalent to this type definition is the following object `Component`, containing a single function `apply`.

```scala
object Component {
  def apply[I, O, S](s: S, i: I): (S, O)
}
```

We can now rewrite `Component` to be an interface and put the state `S` inside `Component` implicitly. This removes the need for an input parameter of type `S` in the `apply` method, since the state is then contained inside the `this` pointer. Instead of the `S` in the return type we now, however, have to return a `Component` that contains the output state.

```scala
trait Component[I, O] {
  // (im)mutable state in here
  def apply(i: I): (Component[I, O], O)
}
```

Instead of returning a tuple, the `apply` method can be split into two separate methods. `update` will accept something of input type `I` and return a new `Component`, containing its new state. The output value of the transformation on the original component together with `I` can be retrieved from the newly returned `Component` using the `action` method.

```scala
trait Component[I, O] {
  // (im)mutable state in here
  def update(i: I): Component[I, O]
  def action: O
}
```

This version of `Component` gives us a first suitable implementation for composing over components. We will refer to this version as the *Immutable Component*. Notice that this version is actually used in our blog post "*Feedback Control for Hackers*" [24] as the basis of the simulation framework used in several case studies.

As an example of a component that inherits this interface, we will show an implementation of a component that maintains a running average of its input elements.

Listing 4.1: Implementation of `RunningAverage` using the *Immutable Component* interface

```scala
class RunningAverage(n: Int, queue: Queue[Double]) extends Component[Double, Double] {
  def update(u: Double): RunningAverage = {
    if (queue.length == n) queue.dequeue
    queue.enqueue(u)

    new RunningAverage(n, queue)
  }

  def action: Double = queue.sum / queue.length
}
```

Here the internal state of the `Component`, that was earlier referred to as `S`, consists of both the integer `n` (representing the number elements over which it has to calculate the running average), and the queue containing at most the latest `n` numbers that it received. Notice that `Component[Double, Double]` here specifies that the input type (the data type it can receive over which it calculates the average) and the output type (the data type of the average) are both `Double`. The `action` method is the one that calculates the actual average using the numbers that are present in the queue. On the other hand, receiving of the next value as well as keeping the queue up to date are done by the `update` method, which returns a new instance of `RunningAverage` with the new state of the queue, including the new element and excluding the oldest element (if the queue's size is equal to `n`).

The observant reader will already have noticed the striking similarity between the *Immutable Component* and the `Component` interface by Janert [28]. In fact, our latest `Component` interface is the immutable variant of his interface. For this, we have to change the output type of the `update` function to `Unit` and perform the action of updating the internal state of the `Component` as a side-effect. The new state will no longer be returned, but is included in the instance of `Component` itself. We will refer to this variant of the interface as the *Mutable Component*.

```scala
trait Component[I, O] {
  // mutable state in here
  def update(i: I): Unit
  def action: O
}
```

Using this *Mutable Component* interface we can implement `RunningAverage` again. Rather than having the `queue` as a constructor parameter, we can now treat it as a field that is automatically instantiation on construction. With this we can mutate the queue when `update` is called. Notice that for this we either need to use a mutable queue here or declare the field mutable by using a `var`. Finally, the implementation of `action` doesn't change with respect to the former version.

Listing 4.2: Implementation of `RunningAverage` using the *Mutable Component* interface

```scala
class RunningAverage(n: Int) extends Component[Double, Double] {
  val queue = Queue[Double]()

  def update(u: Double): Unit = {
    if (queue.length == n) queue.dequeue
    queue.enqueue(u)
  }

  def action: Double = queue.sum / queue.length
}
```

Using the *Mutable Component*, calling the `work` method in Janert's `Component` interface is equivalent to calling `update` and `action` in sequence. To conform to this interface, we can write our `Component` as shown below. In technical terms this is called the coproduct, which we have already seen briefly in Section 1.2.2.

```scala
trait Component[I, O] {
  // mutable state in here
  def update(i: I): O
}
```

As Janert already concluded, this interface is fairly suitable for simulation purposed, but does is not meant for production services. Although it is possible to use this interface in production, it would mean that we would introduce a fair bit of mutable state, which is not the most desirable thing in todays distributed systems, microservice architecture or APIs. Besides that, the implementer of this `Component` interface has to deal with concurrency all by himself if he wants to make sure that no race conditions or other concurrency side effects happen.

Rather than letting the implementer deal with these issues, we think that the interface should be as simple as possible and that the implementer should only have to focus upon the actual behavior of a certain `Component`. All the concurrency, scalability, fault tolerance should be handled by the interface and operations for composing instances of this interface which we will discuss later.

In order to move toward this goal we will again perform a couple of transformations on our `Component` interface. We first of all require *continuation-passing-style*, which transforms a regular function with input A and output B into a function that takes a second argument which is a function from the output type B to C:

$$A \to B \quad \Leftrightarrow \quad (A, B \to C) \to C$$

The new function's second argument is called the continuation, which specifies what to do with the original function's output afterwards. In the code below we define two functions `f` and `g` to be the original function and the continuation-passing-style function respectively. We also define a function `h` which transforms a `B` into a `Unit`. With this we can show that first calling `f` and then `h` gives us the same result as calling `g` with `h` as its second parameter.

```scala
def f(a: A): B
def g(a: A, cont: B => Unit): Unit
def h(b: B): Unit

h(f(a)) == g(a, h)
```

A second transformation that we require is the notion of currying, which is derived from category theory, known as a *Cartesian closed category*. We will not go into the mathematical details, but just focus on the application in the field of programming, types and function. Currying basically means that you can split a list of function parameters into an equivalent series of functions. For example, given a function `f` with parameters of type `A` and `B` and return type `C`, we can define an equivalent function `g` which is the curried form of `f`:

```
def f(a: A, b: B): C
def g(a: A)(b:B): C
```

Function `f` in this example has type `(A, B) ⇒ C`, which is equivalent to the type of `g`: `A ⇒ B ⇒ C`. Notice that the latter is a function with a single argument of type `A` and returns another function with a single argument of type `B` and a return type `C`.

We will use now use continuation passing style and currying in the continuation of our derivation. So far we have an interface which similar to the one presented in Janert's book.

```
trait Component[I, O] {
  // mutable state in here
  def update(i: I): O
}
```

First of all, we can apply continuation passing style and take the output type `O` as the input of a function which will be a second input parameter of the new `update` method.

```
trait Component[I, O] {
  // mutable state in here
  def update(i: I, f: O ⇒ Unit): Unit
}
```

With this new interface we could potentially chain one `Component` to another, by calling the `update` method of the second `Component` in the second input parameter of the first. This however would ugly pretty fast, as lots of components would mean lots of nested functions, which are not very pleasant for the eye. What we will do instead is continue by observing that `update` has two parameters, which means that we can apply currying and decompose the parameter list.

```
trait Component[I, O] {
  // mutable state in here
  def update(i: I)(f: O ⇒ Unit): Unit
}
```

Now that we have a function `update` with type `I ⇒ (O ⇒ Unit) ⇒ Unit`, we can use the right associativity law on functions and conclude that `update` can also be read as a function which has input type `I` and output `(O ⇒ Unit) ⇒ Unit`. Although this return type doesn't seem so useful at first sight, we must remember that this is almost identical to the actual type definition of `Observable` (see Equation (1.2)). The major difference is in the input type of the inner function, which is not `Try[Option[O]]` but just `O` instead. However, it seems perfectly reasonable that the computation inside a `Component` may fail or terminate all computation inside that `Component`. Following this reasoning, we can now write our interface as follows:

```
trait Component[I, O] {
  // mutable state in here
  def update(i: I): Observable[Unit]
}
```

Although one could stop here and develop operations for composing and executing instances of this interface, we will continue with some more transformations in order to further optimize the upcoming API around this interface. We must observe first of all that there is still mutable state involved in this interface, which we have concluded before we want to minimize as much as possible. Secondly, when sequentially composing two instances of this interface together, we have a way for the `OnNext`s of the first instance to go into the second, but we have no possibility to propagate an `OnError` or `OnCompleted` event from the first instance to the second.

In order to achieve these goals, we next apply the inverse of a coproduct, called a `product` in categorical terms, to split the `update` method into two methods `in` and `out`, which accept the input parameter `I` and return the `Observable[O]` respectively.

```
trait Component[I, O] {
  // mutable state in here
  def in(i: I): Unit
  def out: Observable[O]
}
```

Notice that when concatenating multiple instances of `Component`, we only call `out` once while setting up the chain, whereas `in` gets called every time the previous component emits an element. As discussed before, these emissions only involve the OnNext events, since the OnError and OnCompleted events have nowhere to go. Now that we have split update into two separate methods, we can easily add extra methods for these missing events. Rather than doing that directly, however, we can equally well use the official way to add these methods, which is by inheriting Component from Observer, which already contains these three methods all together.

```
trait Component[I, O] extends Observer[I] {
  // mutable state in here
  def out: Observable[O]
}
```

To get this to work, there is a little bit of plumbing to be done. First of all, we need to get the values received in the Observer into the output Observable and meanwhile transform instances of type I into instances of type O. As we do not intent the implementer of the Component interface to override the input or output functions, we have to introduce a new function into the interface in which the actual functionality of the component can be declared: transform(is: Observable[I]): Observable[O]. We also introduce a Subject which receives the input events from the Observer methods. As discussed in Section 1.2.6, a Subject is both an Observer and an Observable, which means that we can implement the output function as applying the new transform function to this Subject.

Listing 4.3: Component interface

```
trait Component[I, O] extends Observer[I] {
  val subject = Subject[I]()
  override val _subscription = subject._subscription

  def transform(is: Observable[I]): Observable[O]
  def asObservable: Observable[O] = transform(subject)

  override def onNext(value: I): Unit = subject.onNext(value)
  override def onError(e: Throwable): Unit = subject.onError(e)
  override def onCompleted(): Unit = subject.onCompleted()
}
```

Having set up the Component interface in this way, comes with an extra benefit. By using the transform method that converts one Observable into another, we can leverage the operators defined on Observable to bring the mutable state of the Component into the sequence of operators. We will demonstrate the implementation of a Component and the usage of mutable state by following up on the earlier example of the running average.

Listing 4.4: Implementation of RunningAverage using the Component interface

```
class RunningAverage(n: Int) extends Component[Double, Double] {

  def transform(input: Observable[Double]): Observable[Double] =
    input.scanLeft(new Queue[Double]) { case (queue, value) =>
        if (queue.length == n)
          queue.dequeue()
        queue += value
      }
      .drop(1)
      .map(queue => queue.sum / queue.size)
}
```

Implementing a component is just as simple as creating a class that extends Component and implementing the transform function. In the case of RunningAverage the input stream contains numbers of which the average needs to be calculated for the last n received elements. The mutable state of the queue is wrapped in the scanLeft operator[2], in which the queue gets updated. Since the scanLeft emits the initial empty queue as its first element, we drop this element before proceeding. Finally we transform the queue into the running average inside the map operation.

Based on this interface we think we can present a good foundation for the interface of this feedback API. The concerns we had with the API by Janert are fixed in this interface. Mutable state is no longer required as the Observable provides operators to deal with them. Concurrency issues are dealt with by the Observable as well, as discussed in earlier chapters. Finally, the Component does not run on an externally defined clock but is triggered when it receives an event.

[2]RxJava/RxScala calls this method scan, whereas other implementations such as Rx.NET and RxMobile call it scanLeft

4.2.2 Operators

As mentioned before we have to define operators on the `Component` interface in order to construct larger, more complex components from smaller components and to finally reach the state in which we can create a feedback loop using this interface. In this section we will explore the operator space, see which operators are useful to be part of this API and describe the way they are implemented.

Creating the component

It cannot go without notice that creating an instance of `Component` is quit heavy lifting (see Listing 4.4). We have to extend from the appropriate interface, come up with a suitable name, specify the input and output types and implement the `transform` function. Although this may seem obvious and IDEs like IntelliJ or Eclipse can do most of this automatically, it still is a significant amount of boilerplate code to have for each `Component` to be created. The real essence of this interface is the `transform` function, which both takes and returns an `Observable`. The rest of the interface can be abstracted away into a static function which creates an anonymous instance of `Component` given the `transform` function (Listing 4.5).

The advantage of having this function and calling it `apply` is that it can be omitted in its usage. A simple component like a running sum (or integral) can we written as simple as `Component[Double, Double](_.scanLeft(0.0)(_ + _))`

In practice, while using this interface, it turned out that not always an argument of type `Observable[I] ⇒ Observable[O]` is needed. In many situations a much simpler function would satisfy our needs. Instead it takes a function of type `I ⇒ O` as argument. This function, called `create`, is then used inside the `apply` function.

On the same note we can have an `identity` component which immediately return its input without performing any transformation.

Finally, we add a function `from`, which ignores the input stream and only emits the items from its argument stream.

Listing 4.5: Various ways to create a `Component`

```scala
object Component {
  def apply[I, O](factory: Observable[I] ⇒ Observable[O]) = new Component[I, O] {
    def transform(input: Observable[I]) = factory(input)
  }

  def create[I, O](func: I ⇒ O) = Component[I, O](_.map(func))

  def identity[T] = Component[T, T](Predef.identity)

  def from[Ignore, O](obs: Observable[O]) = Component[Ignore, O](_ ⇒ obs)
}
```

Chaining two components

One of the most important operators would be to compose two components into a single component. The output of the first component is fed to the second component as its input. The input and output of the composed component are the input and output of the first and second component respectively. As the output of the first component is equal to the input of the second component, it must follow that these have the same type. For a first component `c1: Component[I, O]` we must have a second component `c2: Component[O, Y]` such that the composed component `c1.concat(c2)` has type `Component[I, Y]`.

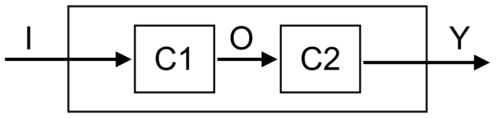


Figure 4.1: Linear composition operator

It should be clear by now that given a value of type `I`, component `c1` should transform this into a value[3] of type `O`. This value is then used as the input of component `c2`, which in turn transforms it into a value of type `Y`. This value is then emitted as the output of the composed component. Besides these `OnNext` events, the `Observable` in either component can return an `OnError` or `OnCompleted` event. In this case the component not only has to propagate these events to the next component, but also needs

[3]Notice that the transformation in `c1` can also cause the output to return multiple values over time or no values at all.

to signal upstream that it is terminating. This means that the `Subject` which is inside the `Component` can unsubscribe itself from every observer, and do this for all upstream components as well.

The behavior for regular `OnNext` events can be achieved by connecting the various streams in Figure 4.1 by subscribing them to each other. Notice that we have to connect the components in reversed order; that is, first subscribe the output of `c2` to the output of the composed component, then subscribe the output of `c1` to the input of `c2` and finally subscribe the input of the composed component to the input of `c1`. The reason for this is that either `c1` or `c2` might have elements that emit immediately when the stream is subscribed. Due to the nature of a `Subject`, these events would get lost if the `Component` did not yet subscribe its output to something else.

The behavior of the terminal events can be achieved by using the functionality of Rx to compose subscriptions. In RxMobile every `Observer` is a `Subscription` as well and likewise every `Component` is an `Observer`, so we just need to *add* the subscriptions to each other. To do so, we also require the subscription that represents the output of the `Component` to be created. We get this by creating a new `Observable` and using the `Observer` in the lambda expression for this.

Listing 4.6: Linear composition operator `concat`

```scala
implicit class Operators[I, O](val src: Component[I, O]) {
  def concat[Y](other: Component[O, Y]): Component[I, Y] = {
    val result = Component[I, Y](input ⇒ Observable.create(output ⇒ {
      other.asObservable.subscribe(output)
      src.asObservable.subscribe(other)
      input.subscribe(src)

      output += other
      other += src
    }))
    src += result
    result
  }
}
```

Abstracting over composition

As we will use this feature of combining the `Subscription` with the `Component` to be created, we will already abstract over this and use this in upcoming operators as well.

Listing 4.7: `lift` operator

```scala
def lift[X, Y](f: (Observable[X], Observer[Y])⇒Unit): Component[X, Y] = {
  val result = Component[X, Y](input ⇒ Observable.create(f(input, _)))
  src += result
  result
}
```

We can then use `lift` in `concat` as follows:

Listing 4.8: Revised implementation of the `concat` operator

```scala
implicit class Operators[I, O](val src: Component[I, O]) {
  def concat[Y](other: Component[O, Y]): Component[I, Y] =
    src.lift((input, output) ⇒ {
      other.asObservable.subscribe(output)
      src.asObservable.subscribe(other)
      input.subscribe(src)

      output += other
      other += src
    })
}
```

In general `lift` can be viewed as the creation of a new, encapsulating `Component`, whose behavior is defined in the function `f`. The function arguments of types `Observable` and `Observer` represent the input and output of this new `Component` respectively. This may seem counterintuitive at first, as a `Component` has an input of type `Observer` and an output of type `Observable`. However, these types are what we see while looking at a `Component` from the outside, whereas within the `lift` operator, we are looking at it from the inside. In that case the input has type `Observable`, since we want to receive new events and the output has type `Observer`, as we want to send new events.

Formalizing the operators

Before we continue developing new operators, we have to observe that both `concat` and `apply`/`create` satisfy the *Arrow* type class that was described first in 2000 in a paper by John Hughes [27]. The *Arrow* type class has similar operators to the `Monad` type class, but is defined more generally. Computations that are clearly not a monad can be composed over in the same way as computations that are a monad using this new type class. Just as a monadic type `M[A]` is representing a computation that delivers an `A`, so an arrow type `A[B, C]` is representing a computation with input `B` and output `C`. With that the dependence on the input of the computation is made explicit.

The *Arrow* type class defines a couple of functions which are listed in Listing 4.9 in Haskell. First of all, it defines `arr` to be a function that accepts a function from `b` to `c` as its input and the arrow from `b` to `c` as its output. This function is completely analogous to the `apply` method described earlier, where `b` and `c` are both of type `Observable`. Besides that, *Arrow* defines the `(>>>)` operator, which composes two arrows together into one single arrow. Here the output of the first arrow is used as an input for the second arrow. This is of course analogue to the `concat` operator we just implemented. We define `(>>>)` as a second way of writing a linear composition.

Listing 4.9: *Arrow* type class

```
class Arrow a where
    arr :: (b → c) → a b c
    (>>>) :: a b c → a c d → a b d
```

Parallel composition

The careful reader may either have noticed that the definition of an arrow as provided in Listing 4.9 is not complete or wonder how to implement an `add` function in arrow style. As Hughes shows, this is simple in a monadic context, but the current functions in the *Arrow* type class do not satisfy the need.

```
add :: Arrow a ⇒ (a b Int) → (a b Int) → (a b Int)
add x y = ...
```

The input type `b` of the resulting arrow of this function needs to supply its value to both inputs of this functions arguments and combine its outputs. This is however not possible with the current `(>>>)` operator we have defined. Hence we require some new operators to be added to the *Arrow* type class. Hughes defines these operators in terms of each other, leaving only one operator open that requires a custom implementation.

Listing 4.10: *Arrow* type class

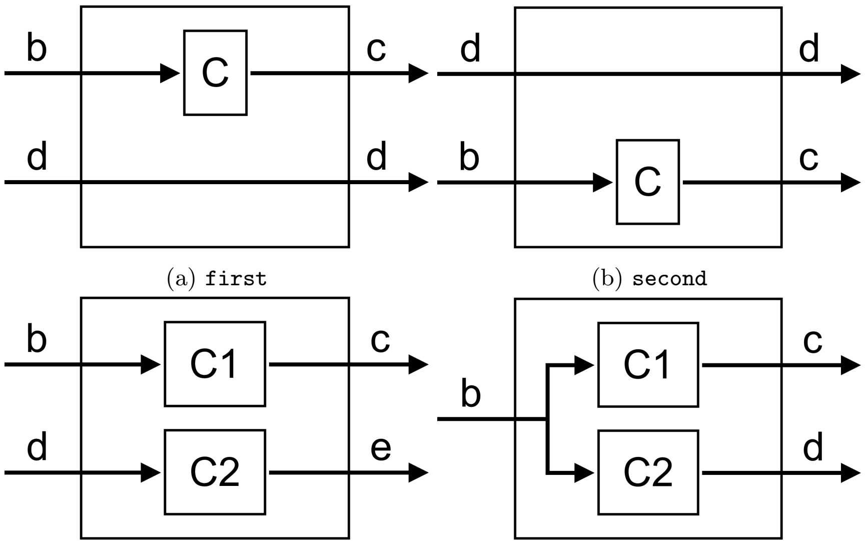
```
class Arrow a where
    arr :: (b → c) → a b c
    (>>>) :: a b c → a c d → a b d
    first :: a b c → a (b,d) (c,d)

    second :: a b c → (d,b) (d,c)
    second f = arr swap >>> first f >>> swap
      where swap (x,y) = (y,x)

    (***) :: a b c → a d e → a (b,d) (c,e)
    f *** g = first f >>> second g

    (&&&) :: a b c → a b d → a b (c,d)
    f &&& g = arr (λb → (b,b)) >>> (f *** g)
```

The operator `first` takes an arrow from `b` to `c` as its argument and just adds an extra input and output to the resulting arrow. `second` does the same, but effectively in reverse, by putting the extra input and output in the first slot. Note how it is possible to implement `second` in terms of `first` by doing extra swaps of the arguments.



(a) `first` (b) `second`

(c) `(***)` or `split` (d) `(&&&)` or `fanout`

Figure 4.2: Parallel composition operators

Implementing these two operators for `Component` requires us to split the '*input tuple stream*', apply the given `Component` to the first part of this stream and merge it back with the second part of the stream. However, remember that for `Observable` there are a number of distinct operators for merging streams, each having their own behavior: `flatMap`, `combineLatest`, `withLatestFrom`, `zip`, `merge`, `concat`, etcetera. Since we want to merge the `Component`'s output stream and the second part of the input stream one-by-one, the `zip` operator is required here. The rest of the implementation details are just straightforward, using the earlier defined `lift` and taking care of the required subscriptions.

Listing 4.11: Implementations of the *Arrow*'s `first` and `second` operators

```scala
implicit class Operators[I, O](val src: Component[I, O]) {
  ...

  def first[Y]: Component[(I, Y), (O, Y)] =
    src.lift((input, downstream) => {
      val srcOut = src.asObservable
      val inputOut = input.map(_._2)

      input.map(_._1).subscribe(src)
      srcOut.zipWithBuffer(inputOut)((_, _)).subscribe(downstream)

      downstream += src
    })

  def second[Y]: Component[(Y, I), (Y, O)] =
    Component.create[(Y, I), (I, Y)](_.swap) >>> src.first >>> Component.create(_.swap)
}
```

The other two operators that Hughes defines for the *Arrow* type class are `(***)` and `(&&&)`. The former operator is used in "*parallel composition*" of arrows, taking its input consisting of two arrows and merging them into one arrow of tuples. This is done by applying `first` to the first input argument and `second` to the second. The latter operator, takes this behavior one step further by specifying that the input type of both input arrows must be the same. With this we can construct a resulting arrow that has an input type `b`, whose values are supplied to both arrows, and an output type which is the tuple (or product) of both arrow's results. This type of composition is sometimes referred to as "*fanout composition*". Implementations of these operators in the context of `Component` are equivalent to the ones in Listing 4.10:

Listing 4.12: Implementations of the *Arrow*'s `(***)` and `(&&&)` operators

```scala
implicit class Operators[I, O](val src: Component[I, O]) {
  ...

  def ***[X, Y](other: Component[X, Y]): Component[(I, X), (O, Y)] =
    src.first[X] >>> other.second[O]

  def &&&[Y](other: Component[I, Y]): Component[I, (O, Y)] =
    Component.create[I, (I, I)](b ⇒ (b, b)) >>> (src *** other)
}
```

Using these new operators, we are now able to define the `add` function in arrow style as described above. We compose the two arrows using a fanout and bring the two results together by using another component that adds the two values together. Notice that this function would be particularly interesting when implementing a PID controller or any other parallel composed set of components.

```haskell
add :: Arrow a ⇒ (a b Int) → (a b Int) → (a b Int)
add x y = (x &&& y) >>> arr (λ(b,c) → b + c)
```

Hughes finishes this trail by generalizing the `add` to a `liftA2`, which is equivalent in functionality to Haskell's `liftM2` which combines the results of two monadic computations. Besides two input arrows, `liftA2` accepts a combinator function that is applied after the fanout composition of the two arrows (see Figure 4.3). We define a similar operator called `combine` in our API for the purpose of parallel composition:

Listing 4.13: Implementation of the `Arrow`'s `liftA2` or `combine` operator

```scala
implicit class Operators[I, O](val src: Component[I, O]) {
  ...

  def combine[X, Y](other: Component[I, X])(comb: (O, X) ⇒ Y): Component[I, Y] =
    (src &&& other) >>> Component.create(comb.tupled)
}
```

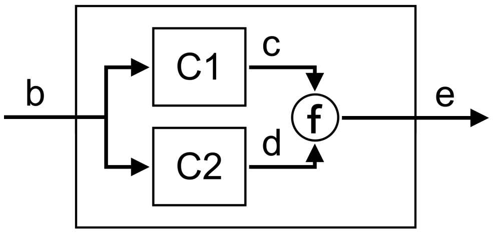


Figure 4.3: Parallel composition

Problems with parallel composition on `Component`

Although our implementation of these parallel composition operators in the context of `Component` might seem fine at first glance, the observant reader might have spotted some flaws in it. The main concern here is the choice of using `zip` in the `first` operator. As discussed in section Section 1.3, `zip` is particularly risky to use given the unpredictable nature of its source streams. This same danger comes up in our implementation of `first`, as the inner component might produce more than one element for every received element or as it might start with a certain number of elements before producing elements based on what it receives[4].

Besides that, given that `zip` is used in the `first` operator, it follows that it is also used in `second` and that it is technically used twice in `(***)`, `(&&&)` and `combine`. This is not only twice as risky, but is also fairly inefficient as the underlying Rx code needs to maintain two queues now. We will show that this can be done more efficiently by reducing the uses of `zip` to only once per `combine` operation.

For this we have to observe that we can not only implement `(***)` in terms of `first` and `second` but that it is also possible to implement both `first` and `second` in terms of `(***)`! After all, `first` is completely equal to `(***)` with the second input arrow being the identity. We can easily see this by

[4]This can be achieved using `startWith`, which is a well-known operator for `Observable` and which we will introduce later in the context of `Component`.

comparing Figure 4.2a to Figure 4.2c where `c2` is equal to `Component.identity`. The same is true for `second` by making `c1` in Figure 4.2c the identity `Component`.

The `(***)` operator can then be implemented in terms of `lift` by splitting the input stream and subscribing both parts to the first and second component respectively. Then the outputs of these components are zipped together into a tuple and subscribed to the output of the resulting `Component`. After that we are only left with some administrative work in terms of the subscriptions.

The other operators, `(&&&)` and `combine` do not require any modifications as they were already implemented in terms of `(***)`. The full implementation of the parallel composition operators in our API now looks as follows:

Listing 4.14: New implementation of the *Arrow*'s operators

```scala
implicit class Operators[I, O](val src: Component[I, O]) {
  ...

  def first[Y]: Component[(I, Y), (O, Y)] = src *** Component.identity[Y]

  def second[Y]: Component[(Y, I), (Y, O)] = Component.identity[Y] *** src

  def ***[X, Y](other: Component[X, Y]): Component[(I, X), (O, Y)] =
    src.lift((input, downstream) => {
      input.map(_._1).subscribe(src)
      input.map(_._2).subscribe(other)

      src.asObservable.zipWithBuffer(other.asObservable)((_, _)).subscribe(downstream)

      downstream += src
      downstream += other
    })

  def &&&[Y](other: Component[I, Y]): Component[I, (O, Y)] =
    Component.create[I, (I, I)](b => (b, b)) >>> (src *** other)

  def combine[X, Y](other: Component[I, X])(comb: (O, X) => Y): Component[I, Y] =
    (src &&& other) >>> Component.create(comb.tupled)
}
```

Functor, Applicative, Monad

One of the reasons that Hughes introduced the *Arrow* [27] is that he saw that not all kinds of computations can be described in terms of monads. He uses the parser library by Swierstra and Duponcheel [39] as an example of a structure that can implement the choice combinator from the *MonadPlus* type class but turns out to have no possible implementation for the *Monad*'s `(>>=)` operator. Their solution to this problem is to discard the use of a *Monad* and instead use a different operator `(<*>)` that later became known as part of the *Applicative* type class. Nowadays *Applicative* sits right in the middle between *Functor* and *Monad* and *MonadPlus* not only 'inherits' from *Monad* but also from *Alternative*, which already has the choice combinator (see Listing 4.16).

As an alternative to introducing the `(<*>)` operator as Swierstra and Duponcheel did, Hughes introduces the *Arrow* as the answer to combining non-monadic computations. With this he shows that every *Monad* is in fact an *Arrow*, but that not every *Arrow* is a *Monad*. For this he introduces another type class *ArrowApply* (Listing 4.15), which he shows to be equivalent to the *Monad* type class. From this it follows that for an *Arrow* to be also a *Monad* it must be able to implement the *ArrowApply* type class.

Listing 4.15: *ArrowApply* type class

```haskell
class Arrow a => ArrowApply a where
  app :: a (a b c, b) c
```

```
class Functor f where
  fmap :: (a → b) → f a → f b

class Functor f ⇒ Applicative f where
  pure :: a → f a
  (<*>) :: f (a → b) → f a → f b

class Applicative m ⇒ Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b

class Applicative f ⇒ Alternative f where
  empty :: f a
  (<|>) :: f a → f a → f a

class (Alternative m, Monad m) ⇒ MonadPlus m where
  mzero :: m a
  mzero = empty

  mplus :: m a → m a → m a
  mplus = (<|>)
```

Listing 4.16: Monadic type classes

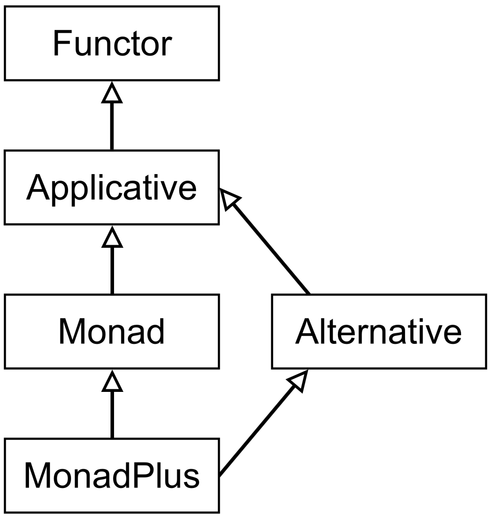


Figure 4.4: Monadic type classes

In the previous sections we have demonstrated the `Component` interface to be an *Arrow*, which, given the discussion above, brings us to the question whether `Component` is also a *Functor*, *Applicative* and *Monad* and hence whether we are allowed to add the associated operators to our API.

The *Functor* type class (Listing 4.16) contains a single function `fmap` which accepts a *Functor* and a function as its input and returns a new *Functor* on which the function is applied. Translated to the `Component` interface, `map`[5] applies a transformation function to the output of the given `Component`. This is simple, as we can already wrap a function in a `Component` using the `create` operator and we can concatenate two components using `concat` or `(>>>)`.

The *Applicative* type class (Listing 4.16) contains two functions, `pure` and `(<*>)`. We already have the former function in our API, namely `from` (Listing 4.5). The latter function is almost the same as `fmap`, only with the difference that the function is wrapped inside an *Applicative*. The two input parameters of `(<*>)` need to be merged into a single *Applicative* where the function from the first parameter is applied to the value from the second parameter. We also already have this functionality in the form of the `combine` operator, which combines two instances of `Component` using a given function.

Listing 4.17: *Functor* and *Applicative* operators

```
implicit class Operators[I, O](val src: Component[I, O]) {
  ...

  def map[X](f: O ⇒ X): Component[I, X] = src >>> Component.create(f)

  def <*>[X, Y](other: Component[I, X])(implicit ev: O <:< (X ⇒ Y)): Component[I,Y] =
    src.combine(other)(ev(_)(_))
}
```

Contrary to the previous type classes, it turns out that we are not able to implement the *Monad*'s `(>>=)` (or `flatMap`) operator. Note that this is equivalent to saying that we are unable to implement the *ArrowApply*'s `app` operator. Writing down these definitions shows how non-sensible both these definitions are:

```
implicit class MonadOperators[I, O](val src: Component[I, O]) {
  def flatMap[X](f: O ⇒ Component[I, X]): Component[I, X]
  def app(implicit ev: I <:< (Component[I, O], I)): Component[I, O]
}
```

From the perspective of `flatMap` it would mean that every time `src` produces an output, it is fed to the function `f`, which would produce a completely new `Component` to be subscribed to the result of `flatMap`. This would mean that after n outputs, there are n instances of `Component` which are 'dynamically' created and that the resulting component will emit n values. The same holds for `app`, which would receive a new `Component` to invoke with every input it receives.

[5]`fmap` in Haskell is usually referred to in Scala as `map`

The fact that `Component` is not a monad may seem surprising at first, but is actually supported by Hughes. In [27] he provides the example of stream processing being an *Arrow* and concludes that this cannot be a fitted into the monadic framework. The *stream processors* in this example are defined as mappings of a stream of inputs to a stream of outputs. Note that this is isomorphic to the `Component` interface. Hughes concludes that "*it turns out that there is no sensible definition of* `app`". "*Since* `app` *would receive a new stream processor to invoke with every input, there is no real sense in which the stream processors it is passed would receive a stream of inputs; we could supply them with only one input each. This would really be very unnatural. Since stream processors do not support a natural definition of* `app`*, they cannot either be fitted into the monadic framework.*"

Reactive operators

Since the `Component` interface is written in terms of Rx, we may also want to use some of the operators that are defined on `Observable`. For example, a components may be a filter or just the functionality that it will skip the first n elements it receives or start by emitting a certain value before responding to the `Component`'s actual input.

One way of doing this would be to create new components for each of these operations as they are needed while writing the feedback system and performing linear composition on these components. However, this would reduce the expressiveness and readability of the program that is written and would mean a lot of repeats of the same code: `x >>> Component(_.`*operator*1`) >>> Component(_.`*operator*2`)`.

A better way of doing this would be to add the most commonly used operators to the `Component` API. By doing so, we can eliminate writing a lot of the same code, make the API more pleasant to use and improve the readability of the code. In fact, we have already introduced one operator that is also an operator in the Rx libraries and follows the implementation pattern as shown above: `map` (Listing 4.17).

Because the implementations of these operators will be almost identical in every case, we first introduce an operator `liftRx` (Listing 4.18), which lifts an Rx operator into the `Component` API such that it can be used in the API as a proper operator. This not only serves the purpose of eliminating duplicate code in the API but can also be used to have a way of lifting an Rx operator that is not present in our API.

Some of the operators present in the API are listed in Listing 4.18, though many more can be thought of to be added.

Listing 4.18: Rx style operators

```scala
implicit class Operators[I, O](val src: Component[I, O]) {
  ...

  def liftRx[Y](f: Observable[O] ⇒ Observable[Y]) = src >>> Component(f)
  def drop(n: Int): Component[I, O] = liftRx(_.drop(n))
  def filter(predicate: O ⇒ Boolean): Component[I, O] = liftRx(_.filter(predicate))
  def map[X](f: O ⇒ X): Component[I, X] = liftRx(_.map(f))
  def sample(interval: Duration, scheduler: Scheduler = NewThreadScheduler()) =
      liftRx(_.sample(interval, scheduler))
  def startWith(o: O): Component[I, O] = liftRx(_.startWith(o))
  def scan[Y](seed: Y)(combiner: (Y, O) ⇒ Y): Component[I, Y] =
      liftRx(_.scanLeft(seed)(combiner))
  def take(n: Int): Component[I, O] = liftRx(_.take(n))
  def throttle(duration: Duration, scheduler: Scheduler = NewThreadScheduler()) =
      liftRx(_.throttle(duration, scheduler))
}
```

Given these new operators, we can reimplement the `RunningAverage` example from Listing 4.4. As all will agree, this looks much better and makes the code more readable and usable!

Listing 4.19: Implementation of `RunningAverage` using the new Rx style operators

```scala
implicit class RunningAverageOperator(val src: Component[Double, Double]) {
  def runningAverage: Component[Double, Double] =
    src.scan(new Queue[Double]) { case (queue, value) ⇒
        if (queue.length == n)
          queue.dequeue()
        queue += value
      }
      .drop(1)
      .map(queue ⇒ queue.sum / queue.size)
}
```

Feedback operator

One of the most needed operations in building feedback systems is *feedback*; the notion of feeding back a `Component`'s output to its input after merging it with the latest element of another stream, while also emitting the same output to the surrounding `Component` (see Figure 4.5).

Though this may seem easy given the infrastructure that we have already build around our API, using `lift` and subscriptions, there are some technicalities that need to be taken into account in this implementation. First of all, in this situation there is a kind of recursion involved on the stream level. This needs some work as the recursed values' emissions need to be postponed to the appropriate moment rather than being emitted directly. Besides that, although the expected behavior of merging a `Component` with `setpoint` can be naively implemented using a `withLatestFrom`, it turns out that this does not work in the case where `setpoint` emits its *only* element before the `Component` emits anything. This has to do with the behavior of `withLatestFrom` which discards everything that is emitted on the second `Observable` as long as the first `Observable` has not yet emitted any values.

The first concern, regarding the recursion, is solvable with the dedicated `TrampolineScheduler` which is available in both RxJava and RxMobile. This scheduler is specially created for recursive streams like this one and "*queues work on the current thread to be executed after the current work completes*" [14]. As long as the stream that runs back into the input of the `Component` is observed on this `Scheduler` (Listing 4.20 line 10), the recursion should work out fine.

The concern about an early emitting `setpoint` is somewhat harder to fix. To solve this, we have to do a `combineLatest` on these streams in order to not discriminate against which stream emits first. However, as soon as both streams have emitted their first value, we have to switch to the original `withLatestFrom` behavior. This is done in the function `loop` in Listing 4.20 line 26. Here the first values of each of the two streams are merged into a tuple using the `combineLatest` and `take(1)`. Notice that the latter causes the stream to complete immediately after the first tuple is emitted. This `OnCompleted` is however postponed by `flatMap` until the inner-stream is completed. Inside the `flatMap` we unpack the tuple again and feed the values as the first new input of the `src` component. Here also the merging behavior is 'replaced' by `withLatestFrom`. Due to our earlier concern with recursion, we have to use the extra `Observable.create` here and feed the first tuple to the `observer` by hand.

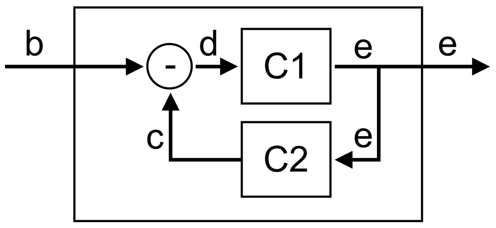


Figure 4.5: Feedback operator

With the `loop` function in place, we can implement the `feedback` operator as shown in Listing 4.20 line 7. As usual we use the `lift` operator as the basis of the implementation. Notice how we have to subscribe the output of `src` twice, once to the output of the encapsulating component (`downstream`) and once to the input of the transducer `trd`. Also notice that the order of subscriptions is important here: we first have to subscribe the output of the transducer to `src` (after applying the `loop` function and observing on the `TrampolineScheduler`) and only then subscribe the `src`'s output to both `downstream` and the transducer. Just as with `concat` this order of subscriptions prevents us from losing initial elements.

Finally note that, since we have to calculate the *tracking error* as the difference between the setpoint and the control output (Equation (3.1)), we have to restrict the `Component`'s input type to be `Numeric`.

Listing 4.20: Feedback operator

```scala
implicit class Operators[I, O](val src: Component[I, O]) {
  ...

  def feedback(f: O => I)(implicit n: Numeric[I]): Component[I, O] =
    feedback(Component.create(f))

  def feedback(tr: Component[O, I])(implicit n: Numeric[I]): Component[I, O] =
    lift((setpoint, downstream) => {
      loop(tr.asObservable, setpoint)((t, s) => n.minus(s, t))
        .observeOn(new TrampolineScheduler)
        .subscribe(src)

      src += src.asObservable
        .publish(source => {
          source.subscribe(downstream)
          source.subscribe(tr)

          source
        }).subscribe()

      downstream += src
      downstream += tr
      tr += src
    })

  private def loop[T, S](transducerOut: Observable[T], setpoint: Observable[S])
      (combinator: (T, S) => I): Observable[I] =
    transducerOut.publish(tos => setpoint.publish(sps =>
      tos.combineLatest(sps)((_, _))
        .take(1)
        .flatMap { case (t, s) =>
          Observable.create[I](observer => {
            tos.withLatestFrom(sps.startWith(s))(combinator).subscribe(observer)
            observer.onNext(combinator(t, s))
          })
        }))
}
```

With this last operator added to the API, we think we have a solid framework for creating and executing feedback systems that can be placed in production worthy systems. The interface as well as the set of operators have their foundations in theoretical concepts such as category theory and functional programming and are built on top of the very successful concept of reactive programming. It is our believe that this API can replace any imperatively created feedback system. We will use our API in Section 4.4 by revisiting the small *ball movement control system* example from Section 3.3.

4.3 Improvements on the API

The reader that is experienced with the concepts of Rx may have spotted an interesting point in the implementations of the operators in the previous section. This already becomes apparent from the implementation of `concat` in Listings 4.6 and 4.8. Although the purpose of this operator is to connect two instances of `Component` in a linear composition, most of the work is spend on administrative work to subscribe streams to each other and add unsubscribe handlers. Also note again that the order in which these subscribe calls are executed is actually important!

One might argue that this is just how this operator is supposed to be set up for it to work correctly. The experienced Rx user will however respond to this by observing that all we are doing is connecting various instances of `Subject` in a somewhat complex manner. This is true, given the implementation of the `Component` interface in Listing 4.3. As derived in Section 4.2.1, a `Component` must be an object that received elements of one type and internally transform these into another type and push out these new elements to those who are subscribed to it. Also we determined in Listing 4.3 that the essence of `Component` is the `transform` function which transforms a stream of elements of one type into a stream of elements of another type. This also becomes apparent in the `apply` function in Listing 4.5, which precisely has this transformation as its argument. The ceremony that is required to make this work in Listing 4.3 involves the `Subject`, `Observer` and the associated subscription management that becomes visible in Listings 4.6 and 4.8.

The refactoring we propose in this section is to remove these ceremonial elements from the `Component` interface and to really make the `transform` function the essence of the interface. For this we remove the `Subject`, the associated `_subscription` value as well as the inheritance from `Observer[I]`. What we end up with is a simple class `Component` with a `transform` function of type `Observable[I] => Observable[O]` as its argument and in single method called `run` that transforms an input stream into an output stream (Listing 4.21). The associated `apply` function is changed accordingly.

Listing 4.21: Revised version of the `Component` interface

```scala
class Component[I, O](transform: Observable[I] => Observable[O]) {

  def run(is: Observable[I]): Observable[O] = transform(is)
}
object Component {
  def apply[I, O](transform: Observable[I] => Observable[O]): Component[I, O] =
    new Component(transform)

  ...
}
```

Due to the way the operators are implemented we are forced to change only three of them, which we will refer to as the primative operators. These are `concat`, `(***)` and `feedback`. All the other operators are composed from these three operators and the `apply` function. The primative operators can be recognized in the previous section as those operators that were implemented in terms of `lift`. With the introduction of this revised version of `Component` we will no longer have a need for `lift` and we can therefore discard this operator. The new implementations of these primitive operators are shown in Listing 4.22

Listing 4.22: Revised implementations of the primitive operators

```scala
implicit class Operators[I, O](val src: Component[I, O]) {
  def concat[X](other: Component[O, X]): Component[I, X] =
    Component(other.run _ compose src.run)

  def ***[X, Y](other: Component[X, Y]): Component[(I, X), (O, Y)] =
    Component(_.publish(ixs => {
      src.run(ixs.map(_._1)).zipWithBuffer(other.run(ixs.map(_._2)))((_, _))
    }))

  def feedback(tr: Component[O, I](implicit n: Numeric[I]): Component[I, O] = {
    import n._

    Component(setpoint => {
      val srcIn = Subject[I]()

      src.run(srcIn)
        .publish(out => {
          loop(tr.run(out), setpoint)((t, s) => s - t)
            .observeOn(new TrampolineScheduler)
            .subscribe(srcIn)

          out
        })
    })
  }
  ...
}
```

Finally we will briefly revisit the issue regarding the `Component` being a *Monad*. This becomes even more interesting since the `Component` is now reduced to a single function. As is known from for example Haskell, the *function* is considered to be a *Monad*:

```
instance Monad ((→) r) where
  return = const
  f >>= k = λr → k (f r) r
```

Therefore we can conclude that also `Component` is actually a *Monad* and we must be able to write an implementation for it. However, as we have argued before, it does not make sense to make `Component` into a *Monad* given its context and the behavior of `flatMap`. Therefore we still reject `flatMap` as an operator in our API.

```
def flatMap[X](f: O ⇒ Component[I, X]): Component[I, X] = {
  Component(_.publish(is ⇒ src.run(is).flatMap(f(_).run(is))))
}
```

The complete and final version of this API can be found in Appendix B.

4.4 Extended example - revisited

To demonstrate the intended usage of our Feedback API, we will return to the toy example developed in section Section 3.3 and refactor the code such that it uses the powers of both our API and Rx most optimally. The goal of this exercise (especially regarding the feedback system in this application) is to recreate Figure 3.4 in terms of the new API and show the close resemblance between the two.

4.4.1 Controller components

First of all, we observe that in this feedback system a PID controller is present, which is, as discussed in Section 3.2.5, one of the most commonly used controllers in feedback systems. It therefore makes sense to create this controller, as well as it's separate components, and let it be part of the library we create in this chapter. This makes this controller better reusable and prevents many different implementations to be created.

Since a PID controller consists of three subcomponents, the proportional, integral and derivative control, it makes sense to first create these operators using our API. For this we will use Equations (3.2), (3.4) and (3.5) as a basis. For simplicity we will only consider integral and derivative control with a fixed dt value in this section, as this case applies to the Ball Movement Control example. The controllers that act on a variable dt value are implemented almost equally, but just with an extra `withLatestFrom` Rx-operator in them.

The proportional controller can be easily written as `Component.create(kp *)`, with `kp` as the proportional controller gain, and is considered too simple to be put in our library.

The integral controller can be described as a running sum over all the data the `Component` receives. This is equivalent to a `scan` operator with a summation operator as the accumulator function. This is shown in Listing 4.23. Due to the fact that we start with an initial element that is not part of the actual data but is just a seed value, we need to discard this first element using a `drop(1)` operation.

The derivative controller is a little more difficult as there does not exist a dedicated operator for this in neither the Rx framework nor in our Feedback API. In the circumstances that we face in this section (fixed dt value), we can approximate the derivative value of a stream of values as the difference between the current and previous element. Therefore we need to use a sliding buffer followed by a pattern match that calculates the difference between them. Notice that buffer operators are not part of our API but are contained in the Rx API and we hence can use them in combination with the `liftRx` operator from our API. The resulting code can be found in Listing 4.23. Note that the `filter` operator is here because the `buffer` operator will produce a list with less values whenever the source stream terminates. This prevents the pattern match inside `map` from failing.

Finally, we can now combine these three controllers into the PI and PID controllers using the `combine` and `<*>` operators.

Listing 4.23: Implementation of the various commonly used controllers

```scala
object Controllers {
  def integralController[T](ki: T)(implicit n: Numeric[T]): Component[T, T] =
    Component.identity[T]
      .scan(n.zero)(n.plus)
      .drop(1)
      .map(n.times(ki, _))

  def derivativeController[T](kd: T)(implicit n: Numeric[T]): Component[T, T] =
    Component.identity[T]
      .startWith(n.zero)
      .liftRx(_.buffer(2, 1))
      .filter(_.size == 2)
      .map {
        case fst :: snd :: Nil => n.minus(snd, fst)
      }
      .map(n.times(kd, _))

  def piController[T](kp: T, ki: T)(implicit n: Numeric[T]): Component[T, T] =
    Component.create(kp *).combine(integralController(ki))(n.plus)

  def pidController[T](kp: T, ki: T, kd: T)(implicit Numeric[T]): Component[T, T] = {
    import n._

    val proportional = Component.create(kp *)
    val integral = integralController(ki)
    val derivative = derivativeController(kd)

    proportional.combine(integral)((p, i) => p + i + _) <*> derivative
  }
}
```

4.4.2 Ball Movement Control - refactored

A next observation to make in our attempts to refactor the Ball Movement Control example is that the code as shown in Listings 3.1 and 3.2 is defined in terms of tuples to distinguish the horizontal and vertical movement. For our new implementation, however, we will separate these dimensions, write a feedback control system that applies to one-dimensional motion and combine two of these systems to get control for two-dimensional motion.

Compared to Listing 3.1, we now not only have to define the position, velocity and acceleration in 2 dimensions, but also in 1 dimension (Listing 4.24). This also holds for the `Ball` class, which now consists of two classes: `Ball1D` and `Ball2D`. The old `Ball.accelerate(Acceleration)`, which transformed an acceleration into a new position, is split into two stages, transforming an acceleration in a new velocity using `AccVel.accelerate` and transforming a velocity into a new position using `Ball1D.move`. Note that these transformations resemble the three arrows in the top middle section of Figure 3.4. Finally we also declare the corresponding *one-dimensional* `BallFeedbackSystem` to be a `Component` that takes positions and emits `Ball1D` instances.

The next step is to translate Figure 3.4 into an actual `BallFeedbackSystem` that is used instead of the imperative version in Listing 3.2. As we already established in the `integral`, a running sum can be established using a `scan` operator with a seed value and an accumulator function. To discard the seed afterwards, `drop(1)` is used. Together with a `map` to have a bound on the acceleration, we can already create the top row of components in Figure 3.4. The `feedback` operator is then used to create the loop, in combination with the extraction of the position from the `Ball1D` object. Finally, since we want a feedback cycle to happen only every 16 milliseconds, we include a `sample` operator right before the `feedback` operator. The full feedback system is shown in Listing 4.25.

Listing 4.24: Ball motion physics

```scala
type Pos = Double
type Vel = Double
type Acc = Double
type Position = (Pos, Pos)
type Velocity = (Vel, Vel)
type Acceleration = (Acc, Acc)
type History = mutable.Queue[Position]
type BallFeedbackSystem = Component[Pos, Ball1D]

case class AccVel(acceleration: Acc = 0.0, velocity: Vel = 0.0) {
  def accelerate(acc: Acc): AccVel = AccVel(acc, velocity + acc)
}

case class Ball1D(acceleration: Acc, velocity: Vel, position: Pos) {
  def move(av: AccVel): Ball1D =
    Ball1D(av.acceleration, av.velocity, position + av.velocity)
}
object Ball1D {
  def apply(position: Pos): Ball1D = Ball1D(0.0, 0.0, position)
}

case class Ball2D(acceleration: Acceleration, velocity: Velocity, position: Position)
object Ball2D {
  def apply(x: Ball1D, y: Ball1D): Ball2D =
    Ball2D((x.acceleration, y.acceleration), (x.velocity, y.velocity), (x.position,
        y.position))
}
```

Note that the drawing of the screen is not done inside the feedback cycle anymore. This is due to the one-dimensionalness of the loop. Also, since this is a side effect that does not have anything to do with the feedback system, we prefer to do this outside the loop. This can be found in Appendix C.

Finally, to create the full two-dimensional system, we combine two instances of the `feedbackSystem` and create a `Ball2D` object as the output type of the full `Component`. Now, given a stream of mouse click events, which acts as a stream of setpoints in regards to the complete system, we can execute this `Component` using the `run` operator.

Listing 4.25: Ball movement feedback system

```scala
val (kp, ki, kd) = (3.0, 0.0001, 80.0)

def feedbackSystem: BallFeedbackSystem =
  Controllers.pidController(kp, ki, kd)
    .map(d => math.max(math.min(d * 0.001, 0.2), -0.2))
    .scan(new AccVel)(_ accelerate _).drop(1)
    .scan(Ball1D(ballRadius))(_ move _)
    .sample(16 milliseconds)
    .feedback(_.position)

def feedback: Component[Position, Ball2D] = {
  val fbcX = Component.create[Position, Pos](_._1) >>> feedbackSystem
  val fbcY = Component.create[Position, Pos](_._2) >>> feedbackSystem

  fbcX.combine(fbcY)(Ball2D(_, _))
}
```

With this example we show how simple it is to create a working feedback system and how strikingly similar the diagram and the corresponding code look. With the API every developer that has some experience with functional and reactive programming can create its own feedback systems and potentially use them to run large-scale production software.

Chapter 5

Solving overproduction with feedback control

In Chapter 2 we discussed how streams can originate from various kinds of sources that can be categorized into three groups. We introduced the *hot* source as a strictly reactive collection of data: there is no way to interact with the stream or control how fast it produces its data. On the contrary, a *cold asynchronous* source can be interacted with, as it has an interface from which one can get zero or more elements. However, as the data to be returned can take some time to be computed, this source is still bound to the same notion of time as is the case with the hot source. Finally there is the group of *cold synchronous* sources, which takes away the notion of time: elements that are requested will be returned immediately.

We also discussed several solutions to overproduction in the light of these three groups of sources. We learned that *avoiding* by grouping or dropping data works perfectly for hot and cold asynchronous sources as a first line of defense. *Callstack blocking* on the other hand is something that is automatically done to cold synchronous sources but can potentially be dangerous to hot and cold asynchronous sources as they might form a buffer of calls on the stack. The *Reactive Streams* solution and RxJava's *reactive pull* are to be used on cold sources alone, and cannot work with a hot source as they go against the contract of reactiveness as defined in [20].

The central problem here is that we want a single reactive interface to share between all kinds of data streams[1]. Note that this already works for hot sources; by definition they are suitable for a reactive interface. We only need a way to interact with cold sources in an overproduction-safe way.

Reactive Streams and RxJava's reactive pull achieve this by introducing the concept of backpressure and changing the reactive interface itself; making the consumer in charge of the data flow, rather than the producer. Not only is this against the concept of reactiveness, it also gives many problems with implementing the operators defined on the reactive interface (see Section 2.2.4).

In this chapter we will propose an alternative to dealing with cold sources, that makes use of the feedback systems theory and API as described in Chapters 3 and 4. We will show how the overproduction problem can be reduced to a control problem that can be solved using a feedback control system. Furthermore we provide a design and implementation for this feedback system and show how this can be fitted in the purely reactive interface that was described in Section 1.2.

5.1 Overview

RxJava points out on its wiki page about backpressure [6] that it does not make the problem of overproduction in the source go away. It claims to only move "*the problem up the chain of operators to a point where it can be handled better*". To do so, they created the *reactive pull* mechanism with operators like `onBackpressureBuffer` and `onBackpressureDrop`, such that the flow control is moved up to these kinds of operators.

We propose to move this flow control even further up the chain; up to the point where the source of the stream is drained in the pipeline of operators. Only there we can have maximum control over how much data is brought into the stream at a particular point in time. With this we do not have the need for infinite buffers as is the case in `onBackpressureBuffer`, nor do we have to drop unprocessed elements as is done with `onBackpressureDrop`. We propose to not wrap the cold source in the `Observable.create` (or any of its derived factory methods) but to wrap it in a universal, interactive interface. This way we are not dependent in our implementation on what kind of source we are dealing with. Given this interactive interface we can fill a *bounded* buffer with as many elements as can be processed at a particular point in

[1]Although one might argue that you ought not to be using a reactive interface for an interactive (cold) source, we acknowledge the fact that in many circumstances it is more practical to view and treat them as 'streaming' and 'real-time' data rather than having them as interactive sources.

time. The buffer pulls data from the source on behalf of the subscriber, which gets as much data pushed at it as it is able to handle. Pushing an element from the buffer to the downstream will automatically block the thread for another element to be pushed until the first one is fully processed.

Since we want to control the buffer's size, we have to observe that control over how many elements will flow into the buffer is to a certain extends in our own hands. However, how fast the downstream is going to drain the buffer is an unknown fact. Therefore we will use feedback control to respond to the uncertainty that the downstream brings in. It does, however, not make sense to give a certain fixed size to the setpoint and compare the current size with it. After all, some 'slow' consumer might go faster or slower than expected. Controlling the buffer with a certain fixed size defeats the purpose of the feedback system in this case, as we cannot dynamically grow or shrink the size as needed. On the other hand, it is also not possible to ask "*make sure the buffer is filled to its optimal size*". A feedback system is not able to solve this, as this does not have a particular setpoint specified [28].

Instead of controlling the buffer size directly, we choose to measure the ratio between what goes out the buffer and what comes in the buffer. We will refer to this ratio as the system's throughput. In an optimal situation the amount of data that comes in is just as much as comes out of the system, so ideally this ratio must be 1.0, which will be the setpoint of this system. Given the error that comes from the difference between the setpoint and the actual throughput, we can then determine how many elements to request from the source in the next iteration. The controller that does this will be discussed in a later section.

The full feedback system is depicted in Figure 5.1. In this figure the *SRC control* receives a number `n` of elements to be requested from source *SRC*. Eventually these elements get delivered and are stored in the buffer inside *Buffer control*. The *Downstream handler* pulls elements from the buffer and sends them downstream to be processed further in the reactive operator sequence. *Buffer control* measures the throughput τ and sends this to the setpoint `sp` to be compared with. The resulting error `e` is then transformed into a new `n` inside an incremental controller C and an integrator Σ.

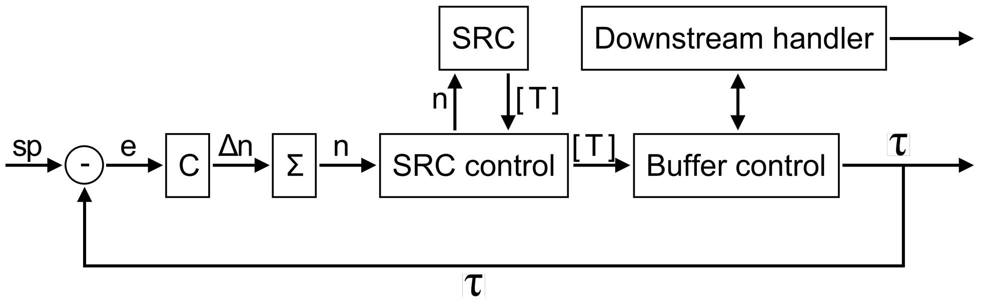


Figure 5.1: Feedback system for controlling overproduction

It becomes clear from this figure that the source itself is not part of the feedback system, but is instead used by the system to retrieve a certain number of elements from. Also note that the *downstream handler* is not part of feedback system either. Even though it *interacts* with the buffer, it is an external force that influences the behavior of the system. Ultimately the *Downstream handler* is the part that exposes an `Observable` for an `Observer` to listen to.

Compared to Reactive Streams and RxJava's reactive pull implementation, our approach has the major advantage of keeping the overflow protection to the only place it can be controlled from: the source of the stream. This allows us to keep the set of operators defined on the reactive interface clean and only have the responsibility of applying a specific functionality to the events that come in. Contrast this with RxJava's backpressure implementation where every operator has to also deal with the question of how many elements it is willing to receive next. This is equal to having our solution of controlling overproduction in every single operator.

5.2 A universal, interactive interface

As mentioned above, we propose to not wrap the (cold) source directly in `Observable.create`, but instead wrap it in a universal, interactive interface. This is necessary since there are many variants of interactive interfaces that all do the same, but each one in a slightly different way.

For example, the `Iterator` interface has an `hasNext` and `next` method, which respectively *check if there is a next element* and *return the next element*. C#'s `IEnumerator` on the contrary has methods such as `moveNext`, which *fetches the next element and returns whether there actually is a next element*, and `current`, which *actually returns the next element*. Java has another interactive collections interface called `Enumeration` that has similar methods. For SQL database interaction, Java defines a `ResultSet`. This interface has a method `next`, which *moves the cursor to the next row of the result*, and methods such as `getInt(int columnIndex)` and `getString(int columnIndex)` to *get the content of a specific type from a column in the row the cursor is pointing to.*

One thing these interfaces have in common is that they contain a method that fetches a single element and in the mean time block the thread it is operating on. If this fetch takes some time, the program will have to wait for the result to come in. To prevent this blocking behavior, we propose a universal interactive interface in which the user requests an element and subscribes to a stream on which *eventually* this element will be emitted. Note that we separate the concerns of *requesting* a next element and *receiving* a next element. In this way, the program can still continue to operate and maybe do some other things while it is waiting for the requested element.

Given that we will use this interface in a feedback system that controls a buffer, we will pose an extra requirement on this interface. As the feedback system's controller might conclude that $n > 1$ elements need to be requested from the source, we must have to possibility to do so. Rather than n times requesting a single element, we want to request n elements at once.

The complete interface is called `Requestable[T]` and is shown in Listing 5.1. It contains a single abstract method `request(n: Int): Unit`, which is called whenever the user of this interface wants a certain number of elements from the source. The requested elements will at some point in time be emitted by the `Observable` that is returned by `results: Observable[T]`. If no more elements are available in the source, this `Observable` will terminate with an `onCompleted` event. The implementor of `Requestable` is expected to use the `subject` to bring elements in the stream, whereas the user of the interface is expected to observe `results` in order to get the requested data. Note that this is a *hot* stream: element emission will not be repeated as a second `Observer` subscribes to the stream.

Example implementations of this interface for `Iterator` and `ResultSet` are included in Appendix D.

Listing 5.1: Universal, interactive interface used in the feedback system

```scala
trait Requestable[T] {
  protected final val subject = PublishSubject[T]()
  final def results: Observable[T] = subject
  def request(n: Int): Unit
}
```

5.3 Controlling overproduction using feedback control

Now that we are able to interact with any cold source via the `Requestable` interface, we can continue designing and discussing the actual feedback system that controls the size of the bounded buffer. As stated before, we do not control the *actual* size of the buffer by using a setpoint of any arbitrary, fixed number of elements. Instead we observe how many elements were taken out of the buffer in relation to how many elements were in the buffer during a particular time span. With that we do in fact not control the buffer's *size*, but rather control the *throughput* of the buffer, while making changes to the number of elements that are requested from the source at every feedback cycle.

The throughput in a particular time span (τ_t) is defined in terms of how many elements are there to be consumed in relation to how many of these elements are actually being consumed. In a scenario where the elements that are not consumed in a certain time span are discarded or where the buffer is flushed at the end of each time span, the throughput would be equal to the ratio of how many elements were being consumed to how many elements were presented to be consumed in a certain time span. In our case, however, we do not wish to discard any elements but rather keep the left-over elements from the previous time span and make them part of what is available to be consumed in the next time span. With this we can define the throughput τ_t at time t as

$$\tau_t = \frac{q_{t-1} + n_t - q_t}{q_{t-1} + n_t} \textbf{ with } q_{t-1},\ q_t,\ n_t \text{ integers} \geq 0 \tag{5.1}$$

or

$$\tau_t = 1 - \frac{q_t}{q_{t-1} + n_t} \textbf{ with } q_{t-1},\ q_t,\ n_t \text{ integers} \geq 0 \tag{5.2}$$

In these formulas, q_t is the size of the buffer at time t, whereas n_t is the number of elements that has been put in the buffer between time $t-1$ and t.

Equation (5.2) provides us with a sense of the range of τ_t. Since $q_t \leq q_{t-1} + n_t$ (it is not possible to take out more elements than are present in the buffer) we can guarantee a lower bound for τ_t of 0.0. Likewise, since $q_{t-1}, q_t, n_t \geq 0$, we can set an upper bound for τ_t of 1.0. Still there is the possibility of dividing by 0, but we will guard against this in the next couple of paragraphs.

$$0.0 \leq \tau_t \leq 1.0 \tag{5.3}$$

With τ as the metric for the feedback system that controls the buffer, it is not difficult to come up with an appropriate setpoint. We want the throughput to be as high as possible, which is, given Equation (5.3), 1.0.

The next point in designing this feedback loop is to determine when a new cycle starts. For this we have to observe that it will only make sense for a new cycle to start if the downstream has polled at least one element from the buffer. If in a certain time span the downstream is too busy processing one element, it does not make any sense to do a new measurement of the throughput. As new elements have been coming in based on the previous feedback cycle, but no elements have been taken out of the buffer, we do not need to request more elements. Instead, we just extend the time span by merging it with the next, until at least one element has been taken out of the buffer. Only then the feedback loop will run a new cycle.

Note that using this definition of a feedback cycle is a guard against dividing by 0 in Equations (5.1) and (5.2). This can only happen when at the start of a time span the buffer is empty and during this time span no elements are coming into the buffer. This can either be due to an unfortunate decision of the controller (which we will discuss in Section 5.4) the request no further elements from the source, even though the buffer is empty, or because it takes some amount of time before the source can produce its next element. If the buffer was empty at the start and no elements were coming in, the downstream would at no point during this time span be able to poll an element from the buffer. Because of this, the current time span is merged with the next time span, without running through a whole new cycle and therefore also without running into dividing by 0 while calculating τ.

Implementation

With this metric and its constraints in mind, we can start implementing this system using the feedback API as described in Chapter 4. For now we will assume the existence of the controller that will be used in this feedback system, even though it will only be discussed first in Section 5.4. We assume a value `controller` of type `Component[Double, Int]`, with an input as the difference between the setpoint value and the actual throughput, and an output as the number of elements to be requested from the source. We furthermore assume the existence of a value `source` of type `Requestable[T]` from which we can request these element of a generic type `T`.

The buffer is modeled as a `BlockingQueue[T]`, such that multiple threads (to put and poll respectively) can safely interact with it. Besides that we introduce two flags of type `AtomicBoolean`, which signal respectively whether the source is completed (it has no more values) and whether there has been a successful poll during the current feedback cycle.

The code for the feedback system itself is shown in Listing 5.2. Given the controller, we first send the number of requested elements to the source, which then starts producing at most this amount of elements. These are received by the feedback system by listening to `source.results`. These elements are then put into the queue.

To measure the throughput of the buffer, we collect the elements during a certain interval. From this we measure how many elements have come in the queue, as well as the total number of elements that are currently in the queue. As a side-effect we reset the flag for `pollerVisited` to false, since we are now done interacting with the queue. Also, we provide a default starting value for the feedback system at this point, since initially the queue was empty and no elements were going in the queue. It is necessary to do so, as we next compare the current situation with the previous situation by using a `buffer(2, 1)`. Finally, we compute the throughput as described in Equation (5.1). This value is fed as the input of the next feedback cycle without performing any operations on the way back.

Listing 5.2: Feedback system for controlling the buffer

```
controller
  .tee(n => source.request(n))
  .liftRx(_.publish(_ => source.results))
  .tee(x => queue.put(x))
  .liftRx(_.buffer(interval.filter(_ => pollerVisited.get())))
  .map(in => (in.size, queue.size))
  .tee(_ => pollerVisited.compareAndSet(true, false))
  .startWith((0, 0)) // initially there is no input and the queue is empty
  .liftRx(_.buffer(2, 1))
  .filter(_.size == 2)
  .map {
    case Seq((_, queueBefore), (in, queueAfter)) =>
      (queueBefore - queueAfter + in).toDouble / (queueBefore + in)
  }
  .feedback(throughput => throughput)
```

The rest of the code, the queue polling behavior, initialization of various values and the wrapping of the whole mechanic in an `Observable.apply`, are considered trivial. This can be found in Appendix D.

5.4 A special controller

The final piece of this feedback control system is the controller, who's job it is to transform the difference between the setpoint and the throughput into a new number of elements to be requested from the source. However, before introduce the controller used in this control system, we first have to observe a number of issues.

We first have to consider the range of error values that is possible within this system. Since we have established that τ_t must be a value between 0.0 and 1.0 (Equation (5.3)), and since we have set the setpoint to a value of 1.0, we must conclude that the range of values for the error must be between 0.0 and 1.0 as well (following Equation (3.1)).

Although this bound may seem to be a good thing, it actually has some interesting implications on the controller of our choice. Janert observes in chapter 15 of his book [28] regarding this kind of bounds that they are not symmetric around the setpoint and that it is not even possible to have a negative error. For a standard PID controller to work well, it should preferably have a range of errors that is symmetric around the setpoint.

Janert suggests to solve this problem by not fixing the setpoint at 1.0 but put it ever so slightly below 1.0. With setpoints like 0.999 or 0.9995, he argues, we will do just as good, as the outcome of the controller will be an integer value rather than a floating point number. We are only able to add or subtract an entire element from the number to be requested. This however causes an unusual asymmetry in the tracking error. Although it can become negative, the error can become much more positive. Using a setpoint of 0.999, the tracking error on the negative side can be at most $0.999 - 1.0 = -0.001$. On the positive side, however, the tracking error can be at most 0.999, which is more than two orders of magnitude larger! As a control action originating from a PID controller is proportional to error, it becomes clear that control actions that tend to increase the number of requested elements will be more than two orders of magnitude stronger than control actions that tend to decrease the number of requested elements. Janert therefore concludes that this is not at all desirable and moves on to a completely new type of controller.

The problem that is discussed in chapter 15 of Janert's book is fairly similar to the situation at hand and so we will create a slightly modified version of his controller. We will keep the setpoint at 1.0, as stated in the previous section. Notice that with this the tracking error can never be negative. Also note that whenever $\tau_t = 1.0$, the tracking error will become zero. This can be interpreted as a signal from the downstream that it was completely able to keep up with the number of elements that were available in the buffer. Most likely this means that the number of requested elements was not high enough for the downstream to be kept busy all the time. We will therefore *increment* the number of requested elements by 1 whenever this happens.

A tracking error greater than zero, on the other hand, signals that the downstream was not able to keep up with the total number of elements that were already present in the buffer and those that were added to the buffer in the previous cycle. We will take an optimistic approach here and assume that this is just an incidental occasion of less elements being consumed. Therefore we change nothing to the number of elements to be requested. We will however keep track of how many times in a row this situation of the tracking error being greater than zero occurs. Only if it happens a certain number of times in a row, we will *decrement* the number of requested elements by 1. From that point on, we will monitor even closer and decrement once again (with briefer periods in between) if the throughput remains less than 1.0. If, however, the throughput comes back to 1.0, we consider it to be a satisfying number of elements, stop decrementing and slowly start increasing the number of requested elements again.

Implementation

As the attentive reader may already have noticed, this is an incremental controller: it does not state how many elements should be requested next, but rather return by how many the number of elements to be requested should be increased or decreased. The actual number is calculated by an extra component added right after the controller, which does the integration over all historical Δn's.

The controller itself is basically a stateful class with a transformation method to construct the next state. Furthermore we have an initial state defined on this class to get everything started. Using the API defined in the previous chapter, we can wrap this state into a `Component` using RxMobile's `Observable.scanLeft` operator. After the newest Δn is extracted from this state, we use a `Component.scanLeft` to compute the actual n. In this step we also prevent the requested number of elements to go negative.

Listing 5.3: Controller implementation for controlling the buffer

```scala
class Controller(time: Int, val change: Int) {
  def handle(error: Double): Controller =
    if (error == 0.0) new Controller(period1, 1) // throughput was 1.0
    else if (time == 1) new Controller(period2, -1)
    else new Controller(time - 1, 0)
}
object Controller {
  def initial = new Controller(period1, 0)
}

val controller = Component[Double, Controller](_.scanLeft(Controller.initial)(_ handle _))
  .drop(1)
  .map(_.change)
  .scanLeft(initialRequest)((sum, d) => scala.math.max(0, sum + d))
```

5.5 An API for cold sources

With the development of a feedback system on buffer control as described in the previous sections, we can create an API that wraps an interactive source in an `Observable`, and let the feedback system draw elements from it based on how fast the downstream is able to handle these elements. To be more precise, given an interactive source, we can wrap it in a `Requestable` as described in Listing 5.1 and use the feedback system in Listing 5.2 to request elements from the source and put them in the buffer. Then the downstream mechanism will poll the buffer continuously and emit these elements in a reactive fashion to a sequence of operators followed by a final `Observer`. With this we can create a function called `from` that takes a `Requestable[T]` as its argument and returns an `Observable[T]` that emits all elements in the source.

```scala
def from[T](source: Requestable[T]): Observable[T]
```

The code for this function mainly consists of a combination of Listings 5.1 to 5.3 and some glue to make it all work together. Refer to Appendix D for the full implementation.

One thing to highlight is that `from` not only takes the wrapped source as an input parameter, but also requires the interval at which the feedback system has to run. This is the `interval` which is used in Listing 5.2 line 5 to determine when to do a next measurement of the throughput in the feedback system.

```scala
def from[T](source: Requestable[T], interval: Duration): Observable[T]
```

As one can imagine, this greatly influences the speed at which the elements are being emitted. If a source can emit elements immediately and the interval is set to 1 second, it will take much longer for all elements to be emitted than when it is set 1 millisecond. Of course the speed will ultimately be determined by how fast the downstream can consume any element. Given the discussion above on when a new feedback cycle is initiated by measuring the throughput, we can conclude that if `interval` is set too fast, it will not influence the performance of the system as a whole, as it will skip the intervals at which the downstream did not show any successful interaction with the queue. If, however, the interval is set at a slower rate than it takes for the downstream to drain the buffer, this will for obvious reasons negatively influence the performance of the system. One could propose to let `interval` be dynamically controlled using another feedback system. However, for now we decided to define it as a constant value in the system and make this part of future research.

With this we have created an API that creates an `Observable` from an interactive source while taking the possibility of overflow into account. Note that we did not have to change or add anything to the Rx interface. Instead we moved the overflow protection to the top of the operator chain, to the point where the source is polled directly. Now, when an `Observer` subscribes to the wrapped cold source, the feedback system will pull from the source on behalf of the `Observer` and push it as a reactive stream to the `Observer`.

To demonstrate the intended use of this API, we included a small example below in Listing 5.4. Here an iterable sequence is created, which is wrapped in a `Requestable.from`. We transform this into a regular `Observable` by calling the `observe` operator with an interval of *1 millisecond*. This operator encapsulates the whole mechanic described in this chapter and will return the `Observable` that polls the buffer. Given this return value, we can now use all operators that are defined on `Observable` and complete the operator chain using a `subscribe`.

Listing 5.4: Using the Requestable API to control the flow of a source

```scala
val sequence: Range = 0 until 133701
val slowConsumer = Observer((i: Int) => println("> " + i),
  e => { e.printStackTrace(); System.exit(0) },
  () => { println("done"); System.exit(0) })

Requestable.from(sequence)
  .observe(1 millisecond)
  .filter(_ % 2 == 0)
  .map(2 *)
  .take(20)
  .subscribe(slowConsumer)
```

Chapter 6

Conclusion

In this final chapter we will reiterate the main contributions of our research, revisit the research questions and discuss both the limitations of our research and future work that arose from this report. The goal of this research is to analyze the problem of overproduction in reactive programming (that is, a producer that produces more data than the consumer can deal with), to examine its currently existing solutions and to come up with a new way to prevent overproduction from happening.

6.1 Contributions

We propose a classification of various sources that can be wrapped in an `Observable`. This classification is based on the behavior of a stream when it is subscribed to by zero, one or multiple `Observers`. A cold stream is known for not doing anything when it has zero subscribers and emitting all elements, right from the start, when it is subscribed to by multiple subscribers. A hot stream, on the other hand, emits its elements whenever they occur. If no subscriber is listening, the data is simply discarded, whereas multiple subscribers that are listening simultaneously receive the same data at (approximately) the same time.

We propose to classify the source of these hot and cold streams in a similar way. A hot stream can only originate from a purely reactive source: it can, following the definition by Berry, not be interacted with and will emit data at its own pace. A classic example of this is a stream of mouse moves, where the consumer is not able to regulate the amount of data that is been sent, because it is not able to communicate with the stream's source. A cold stream, on the other hand, can only originate from a source that can actually be interacted with. In this source, the consumer can determine the speed at which the data is emitted and is thus able to control the source in such a way that it will not produce too much data for the consumer to handle. We further distinguish two types of cold sources: the cold asynchronous source, which after it has received a request for more data might take a while to emit this data and therefore is dependent on a certain notion of time, whereas the cold synchronous source will emit the requested amount of data immediately. An example of the latter source is a list that is already in memory; an example of the former is a SQL query, which may take some time to compute its next record.

A second contribution of this research is an API for constructing and executing feedback control systems that can be used in production level systems. To the best of our knowledge, there are no such libraries yet. The overall problem with feedback control theory is that it is mainly described in terms of mathematical transformations. This is also the case in university classes for computer scientists (if the subject is taught at all). However, these mathematical transformations are currently unknown in the field of computer science and the situations in which we wish to use feedback control. Our API, instead, considers the transformations of which a feedback system consists to be transformations between streams of data that can be composed in several ways and using various operators. We think our API will provide a great tool in designing and executing feedback systems without first having to complete a degree in higher-order mathematics.

Using this new API for feedback control, we are now able to come up with a new solution for preventing overproduction. Since we concluded that the only way in which we can control hot sources is to either buffer or drop data, we mainly focused our solution on cold streams, in which we are able to interact with the source. While other solutions like Reactive Streams and RxJava's reactive pull recursively control for overproduction in every single operator, we propose to only consider overproduction to be a problem at the point at which the source drains its data into the stream. The rationale is that if we control the source's speed at which it drains data into the stream, we do not have the problem of overproduction in the remainder of the stream.

In order to do so, we have created a feedback system around the source, that requests a certain amount of data, which is drained into a buffer. The actual stream, including the operators and consumer, polls data from this buffer for further processing. Based on how much data comes in and goes out of the buffer during a certain interval, the feedback system can then decide to request fewer or more data for the next interval. Given this approach we regulate the flow of data such that the consumer is able to handle every bit of data appropriately.

This solution of controlling the source's speed results in the observation that now it is no longer needed to have any form of flow regulation in every single operator, as is currently the case in RxJava. Given our contribution, the operator implementations no longer requires such things and can have the single responsibility of reactively transforming the data according to the operator's nature.

6.2 Research Questions

In this report we answer three research questions, which will be summarized in this section.

In which way can a reactive program already be controlled to prevent overproduction?

The classification of sources, as described above, brings an important distinction with it: not all currently existing ways of preventing overflow in a stream are suitable for each type of source.

A primitive solution that works for all three types of sources is *buffering*. Here all data that cannot be processed immediately is put in a buffer, such that the consumer can process it at a later time. This solution only works fine for a 'bursty' source, which emits a lot of data at once, followed by a period in which it produces sufficiently less data for the consumer to fully catch up and clear the buffer, before another burst of data comes in. If the behavior of the source is, however, persistent in sending too much data, a buffering solution will not suffice, since the buffer will then grow unboundedly and sooner or later cause the program to run out of memory.

The only solutions that work for a *hot source* are either discarding the data that cannot be processed immediately or grouping it into larger sections and processing it in these groups.

These solutions do also work for the *cold asynchronous sources*, as they are also bound by a certain notion of time. Besides that, the backpressure technique as proposed by Reactive Streams works for these sources as well. This is due to the fact that these sources allow for the consumer to manage the rate at which the producer is allowed to emit its data.

Backpressure also works fine on *cold synchronous sources*, although the best way to control their emission rates is already incorporated at runtime level: callstack blocking. Due to the nature of these streams, the next element can only be emitted once the current element is fully processed. Note that this technique is also naturally used in *hot* and *cold asynchronous sources*, but that this can potentially lead to out-of-memory errors.

How can we implement a *reactive* feedback system that is composed of smaller parts?

A feedback system consists of a number of transformations (or components) that are being composed and where the output of the last transformation is not only the feedback system's output, but is also used as part of the input of the first transformation. The other part of this input is the setpoint, which is combined with the output in any arbitrary way, which serves as the complete input for the first transformation. From this point of view, the whole feedback system is just another component that happens to be constructed out of other, more primative, component. Therefore, the feedback system can potentially be used itself as a component in another feedback system.

We derived an API for such components that builds on top of the RxJava library. We also introduced operators for composing these components in both a sequential and parallel way and related these to the well known structure of *Arrow*s. Finally we introduced a set of operators that makes it possible to turn a sequence of components into a feedback system by feeding back the sequence's output stream to its input stream. The resulting API offers a concise way of describing a feedback system in just a couple of lines of code.

How can the overproduction problem be reduced to a feedback control problem?

We proposed a new solution to the overproduction problem that uses feedback control at its core. While Reactive Stream's backpressure implementation in RxJava moves the overflow control up to a certain operator that can be anywhere in the operator sequence, we chose to move it up to the point where the source is drained into the operator sequence. There we chose to place a buffer and control the number of elements that are being requested from the source. This is done by measuring the *throughput* of the buffer, which is equal to the ratio between how many elements are being pulled out from the buffer and the number of element that were in the buffer during a particular time frame.

With this we have reduced the problem of overproduction to a problem of controlling the size of a buffer. Here the metric (or control output) is the *throughput*, which we bound to be a number between 0.0 and 1.0 (boundaries inclusive). Since the ideal value for the throughput is 1.0 (meaning that all data is consumed directly), we set this as the setpoint of the feedback system that controls the buffer. Using a custom, incremental controller, we transform the tracking error into a new number of elements to be requested from the source.

6.3 Limitations and future work

The proposed solution for controlling overflow does have its limitations. First of all, we were not able to compare our approach with the backpressure solution that is offered by Reactive Streams in implementations such as RxJava's reactive pull in an experimental way. Because we required a *purely reactive* API, we were not able to implement our solution in RxJava directly; otherwise our solution would have implicitly relied on the reactive pull implementation. Instead we decided to use the RxMobile reference implementation that was recently written by Erik Meijer. This implementation is purely reactive, does not take backpressure, reactive pull or any other measures for controlling overflow into account. It instead makes the assumption that the consumer is responsible for making sure it can handle as much data as is served to it, which follows from the definition of a reactive program by Berry. We fully acknowledge that RxMobile is a *reference implementation*, which is not optimized for performance by any means. This is of course in contrast to RxJava, which has performance as one of its main priorities.

Comparing our approach of controlling overflow with RxJava's reactive pull implementation in an experimental way would require us to account for the performance optimizations in RxJava and the lack thereof in RxMobile in our measurements. Currently measuring the performance of both solutions to controlling overflow in any way would be greatly affected by this optimization factor and would therefore give RxJava an unfair advantage over RxMobile. Performing useful experiments to compare both implementations requires us to have equal (or at least similar in terms of performance optimization) implementations on the rest of the API.

Another limitation of our overflow solution is that it only provides an alternative for RxJava's `onBackpressureBuffer`. Each element that is not processed immediately will be buffered for later processing. The alternative to this would be an overflow solution that is equal to RxJava's `onBackpressureDrop` operator, which discards all data that cannot be processed immediately. We think that, given our current solution to `onBackpressureBuffer`, it would not be very hard to modify this and implement a similar solution for `onBackpressureDrop`. Instead of keeping the unprocessed data in a buffer, we would have to clear the buffer after each feedback cycle. This would, however, also require a modification to the metric we use in the feedback loop, as this currently takes into account the size of the buffer. This also gives rise to a new mathematical analysis of this metric.

A further generalization of these different strategies to backpressure is given in the new and upcoming RxJava2.0, where the user is able to, besides using the `onBackpressureDrop` and `onBackpressureBuffer`, implement its own strategy to backpressure. This is a great improvement, which gives the user much more freedom. It would be interesting to investigate a similar generalization in our overflow solution and to so what is still required to make this change.

Besides that, further research is needed to the performance of the controller in our approach of overflow protection. We currently have a very simple controller that changes the requested number of elements by at most one per feedback cycle. Further research is needed to make this controller more responsive and to make better decisions of how many elements are being requested per feedback cycle.

Related to control theory and feedback control we observed earlier in this report that its use is widespread in all areas of science and engineering, except for the field of computer science. Despite having this fairly simple technique to our disposal, we as computer scientists still stick to more complex algorithms. We hope the contribution of our API will spark some interest in this area. We see great opportunities for this technique in many parts of computer science that have to deal with control, such as cloud computing, network control and machine learning.

References

[1] Akka. http://akka.io/. Accessed: 2015-12-28.

[2] Backpressure. https://xgrommx.github.io/rx-book/content/getting_started_with_rxjs/creating_and_querying_observable_sequences/backpressure.html. Accessed: 2015-12-28.

[3] Callstack blocking as a flow control alternative to backpressure. https://github.com/ReactiveX/RxJava/wiki/Backpressure#callstack-blocking-as-a-flow-control-alternative-to-backpressure. Accessed: 2016-1-12.

[4] Elm, the best of functional programming in your browser. http://elm-lang.org/. Accessed: 2015-12-28.

[5] Hot and Cold Observables and multicasted Observables. https://github.com/ReactiveX/RxJava/wiki/Backpressure#hot-and-cold-observables-and-multicasted-observables. Accessed: 2016-01-15.

[6] How a subscriber establishes reactive pull backpressure. https://github.com/ReactiveX/RxJava/wiki/Backpressure#how-a-subscriber-establishes-reactive-pull-backpressure. Accessed: 2015-12-28.

[7] Matlab feedback simulation. http://nl.mathworks.com/help/control/ref/feedback.html. Accessed: 2016-08-13.

[8] Reactive Streams. http://www.reactive-streams.org/. Accessed: 2015-12-28.

[9] Reactive Streams API. http://www.reactive-streams.org/reactive-streams-1.0.0-javadoc/. Accessed: 2015-12-28.

[10] ReactiveX, an API for asynchronous programming with observable streams. http://reactivex.io/. Accessed: 2015-12-28.

[11] Rx on Codeplex. https://rx.codeplex.com/. Accessed: 2016-02-19.

[12] Rx (Reactive Extensions) on Codeplex. https://rx.codeplex.com/. Accessed: 2015-12-28.

[13] RxJava API. http://reactivex.io/RxJava/javadoc/rx/Observable.html. Accessed: 2016-01-04.

[14] RxJava `TrampolineScheduler` documentation. http://reactivex.io/RxJava/javadoc/rx/schedulers/Schedulers.html#trampoline(). Accessed: 2016-08-13.

[15] RxJava scheduling example. http://reactivex.io/documentation/scheduler.html. Accessed: 2016-02-12.

[16] RxJava source code. https://github.com/ReactiveX/RxJava/. Accessed: 2016-01-21.

[17] RxMobile User Guide. To be published later.

[18] Rx Design Guidelines. http://go.microsoft.com/fwlink/?LinkID=205219, October 2010. Accessed: 2015-12-28.

[19] Beckmann, N., and Sanchez, D. Cache calculus: Modeling caches through differential equations.

[20] Berry, G. Real time programming: special purpose or general purpose languages, rapport de recherche inria, n 1065. *11th IFIP World Congress* (1989).

[21] Carroll, J., and Long, D. *Theory of finite automata with an introduction to formal languages.* Prentice Hall, 1989.

[22] Christensen, B. Reactive Programming with Rx. QConSF 2014, http://www.infoq.com/presentations/rx-service-architecture, November 2014. (Combining backpressure operators is discussed at 29:20).

[23] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[24] Heest van, R. Feedback Control for Hackers. http://rvanheest.github.io/Literature-Study-Feedback-Control/, 2015. Accessed: 2016-04-15.

[25] Heest van, R. rvanheest/feedback4s: Release 1.0. https://doi.org/10.5281/zenodo.169095, nov 2016. **doi**:10.5281/zenodo.169095.

[26] Hellerstein, J. L., Diao, Y., Parekh, S., and Tilbury, D. M. *Feedback control of computing systems*. John Wiley & Sons, 2004.

[27] Hughes, J. Generalising monads to arrows. *Science of computer programming 37*, 1 (2000), 67–111.

[28] Janert, P. K. *Feedback control for computer systems*. "O'Reilly Media, Inc.", 2013.

[29] Leshenko, N. Original balltracker implementation by Nikital. https://github.com/nikital/pid, 2015. Accessed: 2016-04-20.

[30] Mealy, G. H. A method for synthesizing sequential circuits. *Bell System Technical Journal 34*, 5 (1955), 1045–1079.

[31] Meijer, E. Subject/Observer is Dual to Iterator. In *FIT: Fun Ideas and Thoughts at the Conference on Programming Language Design and Implementation* (2010).

[32] Meijer, E. Your mouse is a database. *Queue 10*, 3 (2012), 20.

[33] Meijer, E. Duality and the End of Reactive. Lang.NEXT 2014, https://channel9.msdn.com/events/Lang-NEXT/Lang-NEXT-2014/Keynote-Duality, May 2014. Accessed: 2016-01-05.

[34] Meijer, E. Let Me Calculate That For You. Lambda Jam 2014, https://www.youtube.com/watch?v=pOl4E8x3fmw, September 2014. Accessed: 2015-12-28.

[35] Meijer, E., and Beckman, B. Expert to Expert: Brian Beckman and Erik Meijer - Inside the .NET Reactive Framework (Rx). https://channel9.msdn.com/Shows/Going+Deep/Expert-to-Expert-Brian-Beckman-and-Erik-Meijer-Inside-the-NET-Reactive-Framework-Rx, July 2009. Accessed: 2016-01-05.

[36] Meijer, E., and Kapoor, V. The responsive enterprise: Embracing the hacker way. *Communications of the ACM 57*, 12 (2014), 38–43.

[37] Meijer, E., Millikin, K., and Bracha, G. Spicing Up Dart with Side Effects. *Queue 13*, 3 (2015), 40.

[38] Rydeheard, D. E., and Burstall, R. M. *Computational category theory*, vol. 152. Prentice Hall Englewood Cliffs, 1988.

[39] Swierstra, S. D., and Duponcheel, L. Deterministic, error-correcting combinator parsers. In *International School on Advanced Functional Programming* (1996), Springer, pp. 184–207.

[40] Tanenbaum, A. S., and Wetherall, D. J. *Computer Networks*, 5th ed. Prentice Hall Professional Technical Reference, 2011.

Appendices

Appendix A

Ball movement control - Imperative

Listing A.1: Ball movement control

```scala
import javafx.application.Application
import javafx.application.Platform.runLater
import javafx.geometry.Pos
import javafx.scene.canvas.{Canvas, GraphicsContext}
import javafx.scene.input.MouseEvent
import javafx.scene.layout.StackPane
import javafx.scene.paint.Color
import javafx.scene.shape.StrokeLineCap
import javafx.scene.{Scene, SnapshotParameters}
import javafx.stage.{Stage, WindowEvent}

import scala.collection.mutable
import scala.language.postfixOps

class BallMovement extends Application {

  var ball = Ball(ballRadius)
  var setpoint = ball.position
  var prevError, integral = (0.0, 0.0)
  val history = new History
  var historyTick = 0

  def pid: Acceleration = {
    val (kp, ki, kd) = (3.0, 0.0001, 80.0)
    val error = setpoint - ball.position
    val derivative = error - prevError

    integral = integral + error
    prevError = error

    (error * kp + integral * ki + derivative * kd) * 0.001
  }

  def update(implicit gc: GraphicsContext): Unit = {
    val acceleration = pid map (a => math.max(math.min(a, 0.2), -0.2))
    ball = ball accelerate acceleration

    // managing the history
    historyTick += 1
    if (historyTick == 5) {
      historyTick = 0
      if (history.size >= 50)
        history.dequeue
      history enqueue ball.position
    }

    // drawing all the elements
    runLater(() => Draw.draw(ball.position, setpoint, ball.acceleration, history))
  }

  def start(stage: Stage) = {
    val canvas = new Canvas(width, height)
    val root = new StackPane(canvas)
    root setAlignment Pos.TOP_LEFT
    root.addEventHandler(MouseEvent.MOUSE_CLICKED,
        (e: MouseEvent) => setpoint = (e.getX, e.getY))

    implicit val gc = canvas getGraphicsContext2D
    var running = true
    val loop = new Thread(() => while (running) { update; Thread sleep 16 })

    stage setOnHidden ((_: WindowEvent) => { running = false; loop.join() })
    stage setScene new Scene(root, width, height)
    stage setTitle "Balltracker"
    stage show()

    loop start()
  }
}
object BallMovement extends App {
  Application.launch(classOf[BallMovement])
}
```

Listing A.2: Ball movement `Draw` object

```scala
object Draw {

  def draw(pos: Position, setpoint: Position, acc: Acceleration, history:
      History)(implicit gc: GraphicsContext) = {
    drawBackground
    drawHistory(history)
    drawLine(pos, setpoint)
    drawSetpoint(setpoint)
    drawBall(pos)
    drawVectors(pos, acc)
  }

  def drawBackground(implicit gc: GraphicsContext) = {
    gc.setFill(Color.rgb(231, 212, 146))
    gc.fillRect(0, 0, width, height)
  }

  def drawBall(point: Position)(implicit gc: GraphicsContext) = {
    val (x, y) = point
    val diameter = 2 * ballRadius

    gc.setFill(Color.rgb(123, 87, 71))
    gc.fillOval(x - ballRadius, y - ballRadius, diameter, diameter)
  }

  def drawSetpoint(setpoint: Position)(implicit gc: GraphicsContext) = {
    val (x, y) = setpoint

    val radius = ballRadius / 4
    val diameter = radius * 2

    gc.setFill(Color.rgb(161, 90, 90))
    gc.fillOval(x - radius, y - radius, diameter, diameter)
  }

  def drawLine(ball: Position, setpoint: Position)(implicit gc: GraphicsContext) = {
    gc.setStroke(Color.rgb(96, 185, 154))
    gc.setLineWidth(1.0)
    gc.setLineDashes(8.0, 14.0)

    gc.beginPath()
    (gc.moveTo _).tupled(setpoint)
    (gc.lineTo _).tupled(ball)
    gc.stroke()
    gc.setLineDashes()
  }

  def drawVectors(pos: Position, acc: Acceleration)(implicit gc: GraphicsContext) = {
    val (px, py) = pos
    val (ax, ay) = acc

    gc.setStroke(Color.rgb(247, 120, 37))
    gc.setLineWidth(8)
    gc.setLineCap(StrokeLineCap.ROUND)

    gc.beginPath()
    gc.moveTo(px, py)
    gc.lineTo(px - ax * 300, py)
    gc.stroke()

    gc.beginPath()
    gc.lineTo(px, py)
    gc.lineTo(px, py - ay * 300)
    gc.stroke()
  }

  def drawHistory(history: History)(implicit gc: GraphicsContext) =
    history.synchronized {
      history.zipWithIndex.foreach(item => {
        val ((x, y), index) = item
        val size = history.size
        val alpha = (index: Double) / size

        gc.setFill(Color.rgb(96, 185, 154, alpha))
        gc.fillOval(x, y, 10, 10)
      })
    }
}
```

Listing A.3: Ball movement control

```scala
package object ballmovement {

  val ballRadius = 20.0
  val width = 1024
  val height = 768

  type Position = (Double, Double)
  type Velocity = (Double, Double)
  type Acceleration = (Double, Double)
  type History = mutable.Queue[Position]

  implicit def toHandler[T <: Event](action: T => Unit): EventHandler[T] =
    new EventHandler[T] { override def handle(e: T): Unit = action(e) }

  implicit def toRunnable(runnable: () => Unit): Runnable =
    new Runnable { override def run() = runnable() }

  implicit class Tuple2Math[X: Numeric, Y: Numeric](val src: (X, Y)) {
    import Numeric.Implicits._
    def +(other: (X, Y)) = (src._1 + other._1, src._2 + other._2)
    def -(other: (X, Y)) = (src._1 - other._1, src._2 - other._2)
    def *(scalar: X)(implicit ev: Y =:= X) = (scalar * src._1, scalar * src._2)
    def map[Z](f: X => Z)(implicit ev: Y =:= X): (Z, Z) = (f(src._1), f(src._2))
  }

  case class Ball(acc: Acceleration, vel: Velocity, pos: Position) {
    def accelerate(newAcc: Acceleration) = Ball(newAcc, vel + newAcc, pos + vel +
        newAcc)
  }
  object Ball {
    def apply(radius: Double) = Ball((0.0, 0.0), (0.0, 0.0), (radius, radius))
  }
}
```

Appendix B

Feedback API

Listing B.1: Component class

```scala
import applied_duality.reactive.Observable

class Component[I, O](transform: Observable[I] => Observable[O]) {
  def run(is: Observable[I]): Observable[O] = transform(is)
}

object Component {
  def apply[I, O](transform: Observable[I] => Observable[O]): Component[I, O] =
    new Component(transform)

  def create[I, O](f: I => O): Component[I, O] = Component(_ map f)

  def identity[T]: Component[T, T] = Component[T, T](Predef.identity)
}
```

Listing B.2: Operators on Component

```scala
import applied_duality.reactive.schedulers.{NewThreadScheduler, TrampolineScheduler}
import applied_duality.reactive.{Observable, Observer, Scheduler, Subject}

import scala.concurrent.duration.Duration

package object component {

  implicit class ArrowOperators[I, O](val src: Component[I, O]) {
    def >>>[X](other: Component[O, X]): Component[I, X] = this concat other

    def concat[X](other: Component[O, X]): Component[I, X] =
      Component(other.run _ compose src.run)

    def first[X]: Component[(I, X), (O, X)] = this *** Component.identity[X]

    def second[X]: Component[(X, I), (X, O)] = Component.identity[X] *** src

    def ***[X, Y](other: Component[X, Y]): Component[(I, X), (O, Y)] =
      Component(_.publish(ixs => {
        src.run(ixs.map(_._1)).zipWithBuffer(other.run(ixs.map(_._2)))((_, _))
      }))

    def &&&[X](other: Component[I, X]): Component[I, (O, X)] =
      Component.create[I, (I, I)](a => (a, a)) >>> (src *** other)

    def combine[X, Y](other: Component[I, X])(f: (O, X) => Y): Component[I, Y] =
      (src &&& other) >>> Component.create(f.tupled)
  }

```

```scala
implicit class ApplicativeOperators[I, O](val src: Component[I, O]) {
  def map[X](f: O ⇒ X): Component[I, X] = src >>> Component.create(f)

  def <*>[X, Y](other: Component[I, X])(implicit ev: O <:< (X⇒Y)): Component[I,Y] =
    src.combine(other)(ev(_)(_))

  def *>[X](other: Component[I, X]): Component[I, X] =
    src.map[X ⇒ X](_ ⇒ identity) <*> other

  def <*[X](other: Component[I, X]): Component[I, O] =
    src.map[X ⇒ O](o ⇒ _ ⇒ o) <*> other

  def <**>[X](other: Component[I, O ⇒ X]): Component[I, X] =
    other <*> src
}

implicit class RxOperators[I, O](val src: Component[I, O]) {
  def drop(n: Int): Component[I, O] = liftRx(_.drop(n))

  def dropWhile(predicate: O ⇒ Boolean): Component[I, O] =
      liftRx(_.dropWhile(predicate))

  def filter(predicate: O ⇒ Boolean): Component[I, O] = liftRx(_.filter(predicate))

  def liftRx[Y](f: Observable[O] ⇒ Observable[Y]) = src >>> Component(f)

  def sample(interval: Duration, scheduler: Scheduler = NewThreadScheduler()) =
      liftRx(_.sample(interval, scheduler1))

  def startWith(o: O): Component[I, O] = liftRx(_.startWith(o))

  def scan[Y](seed: Y)(combiner: (Y, O) ⇒ Y): Component[I, Y] =
      liftRx(_.scanLeft(seed)(combiner))

  def take(n: Int): Component[I, O] = liftRx(_.take(n))

  def takeUntil(predicate: O ⇒ Boolean): Component[I, O] =
      liftRx(_.takeUntil(predicate))

  def takeWhile(predicate: O ⇒ Boolean): Component[I, O] =
      liftRx(_.takeWhile(predicate))

  def tee(consumer: O ⇒ Unit): Component[I, O] = liftRx(_.tee(consumer))

  def tee(observer: Observer[O]): Component[I, O] = liftRx(_.tee(observer))

  def throttle(duration: Duration, scheduler: Scheduler = NewThreadScheduler()) =
      liftRx(_.throttle(duration, scheduler))
}

implicit class FeedbackOperators[I, O](val src: Component[I, O]) {
  private def loop[T, S](transducerOut: Observable[T], setpoint: Observable[S])
      (combinator: (T, S) ⇒ I): Observable[I] =
    transducerOut.publish(tos ⇒ setpoint.publish(sps ⇒
      tos.combineLatest(sps)((_, _))
        .take(1)
        .flatMap { case (t, s) ⇒
          Observable.create[I](observer ⇒ {
            tos.withLatestFrom(sps.startWith(s))(combinator).subscribe(observer)
            observer.onNext(combinator(t, s))
          })
        }))

    def feedback(trFunc: O ⇒ I)(implicit n: Numeric[I]): Component[I, O] =
      feedback(Component.create(trFunc))

    def feedback(tr: Component[O,I])(implicit n: Numeric[I]): Component[I, O] =
      feedbackWith(tr)((t, s) ⇒ n.minus(s, t))

    def feedbackWith[T, S](transducerFunc: O ⇒ T)(combinatorFunc: (T, S) ⇒ I):
        Component[S, O] =
      feedbackWith(Component.create(trFunc))(combFunc)

    def feedbackWith[T, S](transducer: Component[O, T])(combinatorFunc: (T, S) ⇒ I):
        Component[S, O] =
      Component(setpoint ⇒ {
        val srcIn = Subject[I]()

        src.run(srcIn)
          .publish(out ⇒ {
            loop(transducer.run(out), setpoint)(combinatorFunc)
              .observeOn(new TrampolineScheduler)
              .subscribe(srcIn)

            out
          })
      })
  }
}
```

Appendix C

Ball movement control - Reactive

Listing C.1: Ball movement control

```scala
import javafx.application.Application
import javafx.rx.Events
import javafx.scene.Scene
import javafx.scene.canvas.Canvas
import javafx.scene.layout.StackPane
import javafx.stage.Stage

import applied_duality.reactive.schedulers.JavaFxScheduler
import fbc.Component
import fbc.commons.Controllers

import scala.concurrent.duration.DurationInt
import scala.language.postfixOps

class BallTracker extends Application {

  val (kp, ki, kd) = (3.0, 0.0001, 80.0)

  def feedback: Component[Position, Ball2D] = {
    val fbcX = Component.create[Position, Pos](_._1) >>> feedbackSystem
    val fbcY = Component.create[Position, Pos](_._2) >>> feedbackSystem

    fbcX.combine(fbcY)(Ball2D(_, _))
  }

  def feedbackSystem: BallFeedbackSystem =
    Controllers.pidController(kp, ki, kd)
      .map(d => math.max(math.min(d * 0.001, 0.2), -0.2))
      .scan(new AccVel)(_ accelerate _).drop(1)
      .scan(Ball1D(ballRadius))(_ move _)
      .sample(16 milliseconds)
      .feedback(_.position)

  def start(stage: Stage) = {
    val canvas = new Canvas(width, height)
    implicit val gc = canvas.getGraphicsContext2D
    Draw.drawInit

    val root = new StackPane(canvas)
    root setAlignment javafx.geometry.Pos.TOP_LEFT

    val history = new History

    root.mouseClicked
      .map(event => (event.getX, event.getY))
      .publish(clicks => feedback.run(clicks).withLatestFrom(clicks)((_, _)))
      .observeOn(JavaFxScheduler())
      .tee(x => {
        val (ball, goal) = x
        Draw.draw(ball.position, goal, ball.acceleration, history)
      })
      .map(_._1.position)
      .buffer(5)
      .map(_.last)
      .subscribe(pos => {
        history.synchronized {
          if (history.size >= 50)
            history dequeue()
          history enqueue pos
        }
      })

    stage setScene new Scene(root, width, height)
    stage setTitle "Balltracker"
    stage show()
  }
}

object BallTracker extends App {
  Application.launch(classOf[BallTracker])
}
```

Listing C.2: Ball movement `Draw` object

```scala
import javafx.scene.canvas.GraphicsContext
import javafx.scene.paint.Color
import javafx.scene.shape.StrokeLineCap

object Draw {

  def draw(pos: Position, setpoint: Position, acc: Acceleration, history:
      History)(implicit gc: GraphicsContext) = {
    drawBackground
    drawHistory(history)
    drawLine(pos, setpoint)
    drawSetpoint(setpoint)
    drawBall(pos)
    drawVectors(pos, acc)
  }

  def drawInit(implicit gc: GraphicsContext) = {
    drawBackground
    drawBall(ballRadius, ballRadius)
  }

  def drawBackground(implicit gc: GraphicsContext) = {
    gc.setFill(Color.rgb(231, 212, 146))
    gc.fillRect(0, 0, width, height)
  }

  def drawBall(point: Position)(implicit gc: GraphicsContext) = {
    val (x, y) = point
    val diameter = 2 * ballRadius

    gc.setFill(Color.rgb(123, 87, 71))
    gc.fillOval(x - ballRadius, y - ballRadius, diameter, diameter)
  }

  def drawSetpoint(setpoint: Position)(implicit gc: GraphicsContext) = {
    val (x, y) = setpoint

    val radius = ballRadius / 4
    val diameter = radius * 2

    gc.setFill(Color.rgb(161, 90, 90))
    gc.fillOval(x - radius, y - radius, diameter, diameter)
  }

  def drawLine(ball: Position, setpoint: Position)(implicit gc: GraphicsContext) = {
    gc.setStroke(Color.rgb(96, 185, 154))
    gc.setLineWidth(1.0)
    gc.setLineDashes(8.0, 14.0)

    gc.beginPath()
    (gc.moveTo _).tupled(setpoint)
    (gc.lineTo _).tupled(ball)
    gc.stroke()
    gc.setLineDashes()
  }

  def drawVectors(pos: Position, acc: Acceleration)(implicit gc: GraphicsContext) = {
    val (px, py) = pos
    val (ax, ay) = acc

    gc.setStroke(Color.rgb(247, 120, 37))
    gc.setLineWidth(8)
    gc.setLineCap(StrokeLineCap.ROUND)

    gc.beginPath()
    gc.moveTo(px, py)
    gc.lineTo(px - ax * 300, py)
    gc.stroke()

    gc.beginPath()
    gc.lineTo(px, py)
    gc.lineTo(px, py - ay * 300)
    gc.stroke()
  }

  def drawHistory(history: History)(implicit gc: GraphicsContext) =
    history.synchronized {
      history.zipWithIndex.foreach(item => {
        val ((x, y), index) = item
        val size = history.size
        val alpha = (index: Double) / size

        gc.setFill(Color.rgb(96, 185, 154, alpha))
        gc.fillOval(x, y, 10, 10)
      })
    }
}
```

Listing C.3: Ball movement control

```
import scala.collection.mutable

package object balltracker {

  val ballRadius = 20.0
  val width = 1024
  val height = 768

  type Pos = Double
  type Vel = Double
  type Acc = Double
  type Position = (Pos, Pos)
  type Velocity = (Vel, Vel)
  type Acceleration = (Acc, Acc)
  type History = mutable.Queue[Position]
  type BallFeedbackSystem = Component[Pos, Ball1D]

  case class AccVel(acceleration: Acc = 0.0, velocity: Vel = 0.0) {
    def accelerate(acc: Acc): AccVel = AccVel(acc, velocity + acc)
  }

  case class Ball1D(acceleration: Acc, velocity: Vel, position: Pos) {
    def move(av: AccVel): Ball1D =
      Ball1D(av.acceleration, av.velocity, position + av.velocity)
  }
  object Ball1D {
    def apply(position: Pos): Ball1D = Ball1D(0.0, 0.0, position)
  }

  case class Ball2D(acceleration: Acceleration, velocity: Velocity, position:
      Position)
  object Ball2D {
    def apply(x: Ball1D, y: Ball1D): Ball2D =
      Ball2D((x.acceleration, y.acceleration), (x.velocity, y.velocity), (x.position,
          y.position))
  }
}
```

Listing C.4: JavaFx/Rx interface

```
package object rx {

  implicit def toHandler[T <: Event](action: T => Unit): EventHandler[T] =
    new EventHandler[T] { override def handle(e: T): Unit = action(e) }

  implicit class Events(val node: Node) extends AnyVal {
    def getEvent[T <: InputEvent](event: EventType[T]): Observable[T] =
        Observable.create[T](observer => {
      val handler = (e: T) => observer.onNext(e)
      node.addEventHandler(event, handler)
      observer += Subscription { node.removeEventHandler(event, handler) }
    })

    def mouseClicked: Observable[MouseEvent] = getEvent(MouseEvent.MOUSE_CLICKED)
  }
}
```

Appendix D

Overproduction solution with feedback control

Listing D.1: Universal, interactive interface

```scala
import java.sql.ResultSet

import applied_duality.reactive.{Observable, Subject}

import scala.concurrent.duration.{Duration, _}
import scala.language.postfixOps

trait Requestable[T] {

  protected final val subject = Subject[T]()

  final def results: Observable[T] = subject

  def request(n: Int): Unit
}
object Requestable {

  implicit class RequestableObservableOperator[T](val requestable: Requestable[T])
      extends AnyVal {
    def observe(timeout: Duration = 1 second): Observable[T] =
      RequestableObservable.from(requestable, timeout)
  }

  def from[T](iterable: Iterable[T]): Requestable[T] = {
    val iterator = iterable.iterator

    new Requestable[T] {
      def request(n: Int): Unit = {
        (0 until n).toStream
          .takeWhile(_ => iterator.hasNext)
          .map(_ => iterator.next())
          .foreach(subject.onNext)

        if (!iterator.hasNext)
          subject.onCompleted()
      }
    }
  }

  def from[T](resultSet: ResultSet)(composer: ResultSet => T) =
    new Requestable[T] {
      def request(n: Int): Unit = {
        (0 until n).toStream
          .takeWhile(_ => resultSet.next())
          .map(_ => composer(resultSet))
          .foreach(subject.onNext)

        if (!resultSet.next())
          subject.onCompleted()
      }
    }
}
```

Listing D.2: Implementation of RequestableObservable

```
import java.util.concurrent.atomic.AtomicBoolean
import java.util.concurrent.{BlockingQueue, LinkedBlockingQueue}

import applied_duality.reactive.{Observable, Subject}
import fbc2.Component

import scala.concurrent.duration._
import scala.language.postfixOps

object RequestableObservable {

  def from[T](source: Requestable[T], intervalDuration: Duration = 1 second):
      Observable[T] =
    Observable.create[T](subscriber => {

      val (period1, period2) = (2, 1)
      val initialRequest = 2

      val queue: BlockingQueue[T] = new LinkedBlockingQueue[T]()
      val upstreamCompleted: AtomicBoolean = new AtomicBoolean(false)
      val pollerVisited: AtomicBoolean = new AtomicBoolean(false)

      val interval = Observable.interval(0 seconds, intervalDuration)

      val input = Subject[Double]()

      class Controller(time: Int, val change: Int) {
        def handle(error: Double): Controller = {
          if (error == 0.0) new Controller(period1, 1) // throughput was 1.0
          else if (time == 1) new Controller(period2, -1)
          else new Controller(time - 1, 0)
        }
      }
      object Controller {
        def initial = new Controller(period1, 0)
      }

      val controller = Component[Double, Controller](_.scanLeft(Controller.initial)(_
          handle _))
        .drop(1)
        .map(_.change)
        .scanLeft(initialRequest)((sum, d) => scala.math.max(0, sum + d))

      val feedbackSubscription = controller
        // source:
        .tee(n => source.request(n))
        .liftRx(_.publish(_ => source.results))
        // buffer
        .tee(x => queue.put(x))
        .liftRx(_.buffer(interval.filter(_ => pollerVisited.get())))
        .map(in => (in.size, queue.size))
        .tee(_ => pollerVisited.compareAndSet(true, false))
        .startWith((0, 0)) // initially there is no input and the queue is empty
        .liftRx(_.buffer(2, 1))
        .filter(_.size == 2)
        .map {
          case Seq((_, queueBefore), (in, queueAfter)) ⇒
            (queueBefore - queueAfter + in).toDouble / (queueBefore + in)
        }
        .feedback(throughput ⇒ throughput)
        .run(input)
        .subscribe(
          d ⇒ {},
          subscriber.onError,
          () ⇒ upstreamCompleted.compareAndSet(false, true))

      input.onNext(1.0)
      subscriber += feedbackSubscription

      try {
        while (!subscriber.isUnsubscribed && !(queue.isEmpty &&
            upstreamCompleted.get())) {
          Option(queue.poll())
            .foreach(t ⇒ {
              pollerVisited.compareAndSet(false, true)
              subscriber.onNext(t)
            })
        }
        subscriber.onCompleted()
      }
      catch {
        case e: Throwable ⇒ subscriber.onError(e)
      }
    })
}
```