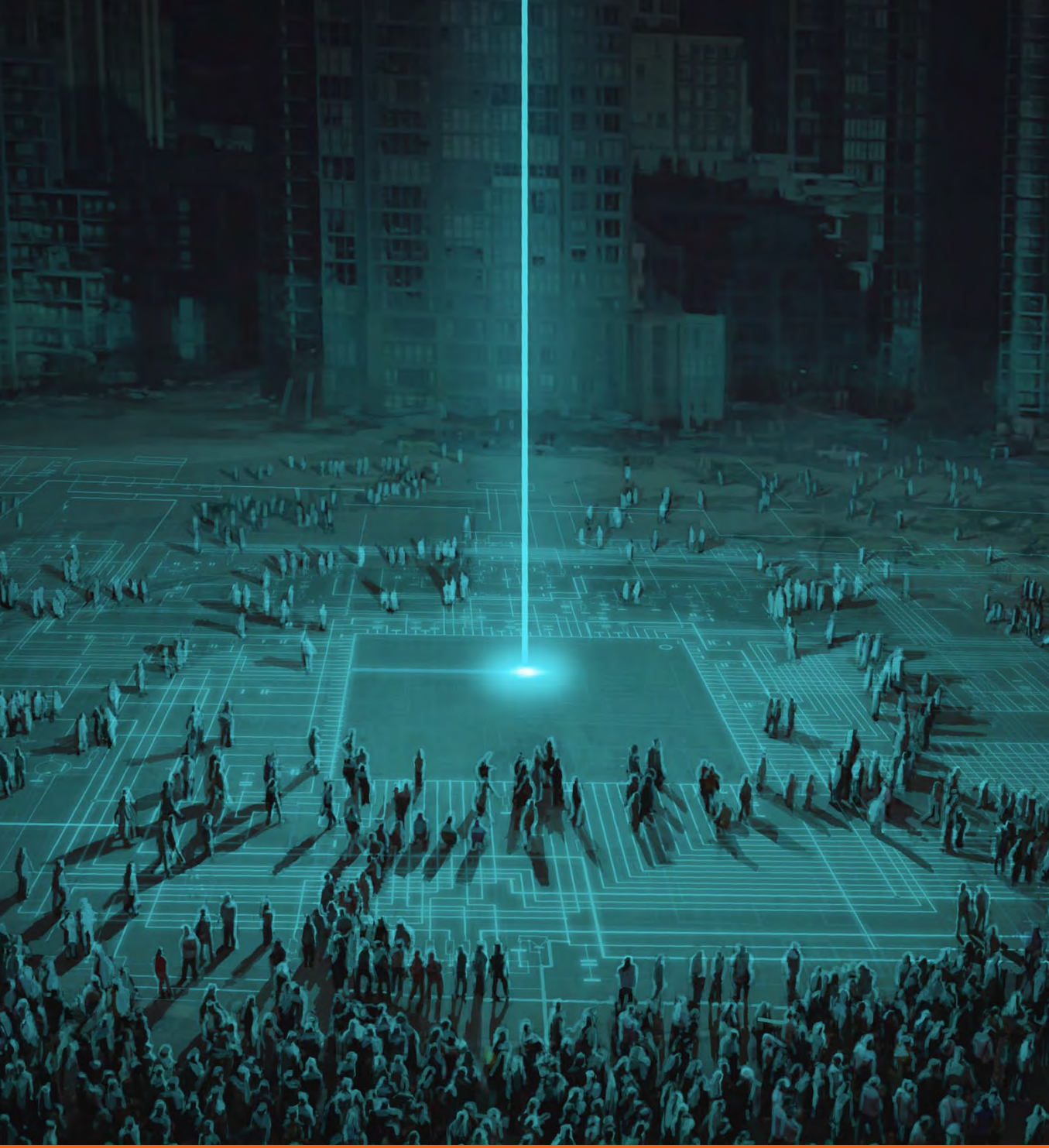
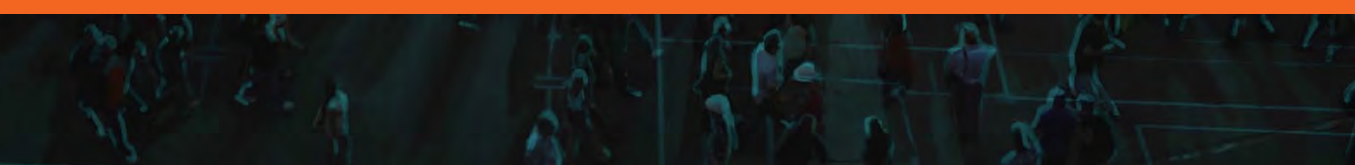


HACKERMONTHLY

Issue 54 Nov 2014



HACK ON YOUR SEARCH ENGINE

and help change the future of search

duckduckhack.com

Curator
Lim Cheng Soon

Contributors
Willem van der Jagt
Daniel Tenner
Andrew Wulf
Eric Adler
André Staltz
Justin Brower

Illustrator
Matus Garaj

Proofreader
Emily Griffin

Printer
Blurb

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be "anything that gratifies one's intellectual curiosity." Every month, we select from the top voted articles on Hacker News and print them in magazine format.
For more, visit *hackermonthly.com*

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

Published by
Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.

Cover Illustration: Matus Garaj

Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-54*

How I Built an Audio Book Reader for My Nearly Blind Grandfather

By WILLEM VAN DER JAGT

Last year, when visiting my family back home in Holland, I also stopped by my grandparents. My grandfather, now 93 years old, had always been a very active man. However, during the preceding couple of months, he'd gone almost completely blind and now spent his days sitting in a chair. Trying to think of something for him to do, I suggested he try out audio books. After finally convincing him -- he said audio books were for sad old people -- that listening to a well performed recording is actually a wonderful experience, I realized the problem of this idea.

The problem with audio devices and the newly blind.

After my first impulse to jump up and go buy him an iPod Touch, I soon realized that, to use an iPod, or any audio device for that matter, one needs to be able to see the tiny controls. So I started looking at existing audio book solutions for the blind. A couple of things exist, but this market seems to be mainly targeted at people that still have a whole life of being blind ahead of them and are willing to invest time into learning very specific technologies. However, this was not my grandfather's situation. I worried that he would lose his motivation (of which he didn't have much left anyway at that point), so I needed to come up with something better. And since I hadn't found anything suitable that I could go out and buy, I would need to build it myself.

Requirements

First of all, of course, whatever I was going to build needed to have an interface that didn't require (much) vision. Second, the controls needed to be intuitive and not require learning any completely new concepts. And last, if my grandfather paused a book, for however long, it would need to continue where he left off, even if the player had been without power.

I will describe in more detail below, but I ended up building a player that used my grandfather's very limited vision. However, it could easily be adapted for someone able to read braille. The player is built using a box the size of a 3 or 4 DVD boxes stacked on top of each other. Each audio book that is stored on the reader has a corresponding DVD box with the title of the book printed in very large letters on the front. When a "book" is placed on top of the reader, the reader starts playing the book. The reader has four large, bright colored buttons on the front with the following functions: pause, rewind twenty seconds, and two buttons that control volume.

The used technologies

Raspberry Pi

At the heart of the player is a Raspberry Pi running Debian Wheezy. Getting Linux to play audio is very easy, so getting to audio books to play wasn't that much of a challenge. For playing audio, I used mpd [musicpd.org], which is a daemon that runs a server that plays audio and that is controlled by sending it commands over TCP, a very reliable and easy to use network protocol.

What makes the Raspberry Pi interesting is not only that it's a tiny computer that runs Linux, but also that it has lots of I/O pins let you connect anything you can imagine (buttons, LEDs, but also serial communication devices). When writing a program for the Raspberry Pi, you'll be able to read from these pins and change the behavior of your program accordingly. The small program I wrote to control the audio book player uses these pins to know when one of the buttons is pressed, and to know which book is placed on the reader. Based on these inputs, it communicates with the mpd server to start, stop, change book etc. etc.

RFID

Each of the DVD boxes that corresponds to one book, contains an RFID card. To read these cards, I connected an RFID card reader to one of the I/O pins that is able to do serial communication so my program knows which book to play. Each RFID card has its

own unique ID, and each audio book is a series of MP3 files that have names starting with this ID.

Getting the books on the reader

I built the reader when I was back in Montreal (which is where I moved from Holland). When I finished the reader, I loaded it with ten books and sent it to my brother who lives a ten-minute walk from my grandparents. My brother took it to my grandfather and explained how it worked. Every time my grandfather finishes his books, my brother takes the reader home with him to connect it to his router. The reader, when powered up, will check for an internet connection, and if it finds one, it sends a message to my phone using Pushover [pushover.net] containing my brother's IP address. I then connect to the reader from my laptop over SSH and copy new books to it.

TECHNICAL DETAILS

1 Playing MP3 Files

Some initial configs

When I ordered my Raspberry Pi (RPi), I also got an SD card already containing Raspbian Wheezy, a Debian port for the Raspberry Pi, which made the setup extremely easy. A couple of important configurations need to be done, though. You can do them on first boot, or skip them because you can't wait to play. In this case you can always go back to the configuration tool by running `raspi-confi`. The most important options to set are `expand_rootfs` and `ssh`. The first expands the partition to use the full SD card, and the second enables an ssh server, so we can access the RPi through ssh later on. SSH is very useful if you don't want RPi running on your TV all the time. Something else I wanted to get out of the way was the login prompt on startup, which my grandfather would never have to use (he doesn't even need to know there is a computer inside his player). Auto login behavior is accomplished by modifying the system's inittab file, located at `/etc/inittab`. Later I actually realized that, to automatically start a script on boot, you don't need to be logged in, so if you're following along, you can skip this. But for completeness, and because you may want this behavior for something else, I will include it anyway.

I commented out the following line:

```
1:2345:respawn:/sbin/getty
--noclear 38400 tty1
```

and put this in its place:

```
1:2345:respawn:/bin/login -f pi
tty1 /dev/tty1 2>&1
```

After rebooting the RPi (`sudo reboot`) I was logged in automatically.

Playing MP3 files

For this project, I would need to control MP3 files from my python code. I had never done this before, and the first thing I found that came close to what I needed was the mixer module that's part of the pygame library. I won't bore you with the code I tried, because I didn't end up using it. I still don't know why, but in my version of pygame, the `pygame.mixer.music.set_pos()` method didn't exist. I briefly verified in the source, but I couldn't even find a reference to it. Since I wasn't even sure pygame was the best option, I continued my search and found the very awesome MPD (music player daemon), which is a daemon that runs a server that plays audio and is controlled by sending it commands over TCP. It runs really well on the Raspberry Pi. It can be easily installed (run `sudo apt-get update` first if this is your first interaction with `apt-get`) :

```
sudo apt-get install mpd
```

The Python client python-mpd [hn.my/pythonmpd] can be installed in any way you prefer. Instructions are in the GitHub repository. MPD is a daemon that accepts connections over TCP on a port (6600 by default, which was fine for me) and are controlled by sending it control strings. By using the python client, I didn't need to worry about formatting the strings and sending them.

MPD doesn't allow us to just play audio files from any location (not that I know of, anyway). You need to give it a location where it will look for them. The installation we just did created a global config file at `/etc/mpd.conf`. The only setting we really care about for now is where to place the audio files. I changed the default setting (which pointed to `/var/lib/mpd/music`) to a folder called `books` in my user folder (`/home/pi/books`). For this change to take effect you need to restart the daemon (`sudo /etc/init.d/mpd restart`).

MPD is now running exactly how I want, and playing audio files from Python becomes really easy. To simply play a file "sometestfile.mp3", which should of course exist in our newly created books folder, could be done as follows:

```
from mpd import MPDClient

client = MPDClient()
```

```
# instantiate the client object
client.connect(host="localhost",
port=6600)
# connect to the mpd daemon
client.update()
# update the mpd database with
# the files in our books folder
client.add("sometestfile.mp3")
# add the file to the playlist
client.play() # play the playlist
```

There's a lot more MPD can do, but we'll get to that when we get to writing the actual code for the book player.

Audio through 3.5mm jack

One last detail before we continue. By default the RPi sends audio over HDMI, and not to the 3.5mm jack I plan to use. I read somewhere that actually by default it detects where it should send it, and at the time of testing, mine was connected to a TV, so that's where it sent it. But I wanted to make sure it didn't automatically send it to HDMI by mistake when my grandfather got the player, so I found out how to configure the built-in audio mixer to always send audio to the analog headphone output (`cset` is to set a configuration variable, we're setting configuration number 3, which is the playback route, to 1, which is the analog out):

```
sudo amixer cset numid=3 1
```

PulseAudio

I found that the playback quality of the RPi when using the standard ALSA sound driver that comes with the Raspbian distribution was pretty good. It did however have one nasty habit: it would generate a loud sharp pop whenever playback was paused. Installing PulseAudio solved this problem, see here for instructions. [hn.my/pulseaudio]

2 Reading RFID Cards

Connecting the RFID reader

I ordered the cheapest RFID reader I could find, because they basically all do the same thing: read an RFID tag and transmit the ID of the card over a serial signal. Some readers offer NFC capabilities (allowing you to store a small amount of data on the card), but I didn't need that. I got mine from *robotshop.com*. It's an Electronic Brick from Seeedstudio. I couldn't find anything on how to connect it to a RPi, but since it communicates over standard UART, I assumed it couldn't be that hard if I just used the Python serial library. [pyserial.sourceforge.net]

Voltage

Even though an RFID device outputs a pretty standard serial signal, something you really need to keep in mind if you don't want to damage your RPi, is that the voltage that the RFID reader outputs on the Tx (the transmitting pin) is

5 volts, and the Rx (the receiving pin) on the RPi only expects 3.3 volts. Connecting this RFID reader directly to the RPi would burn out the Rx pin in the best case. To bring the voltage down to 3.3 volts, I got a logic level converter. Hooking it up to the RPi and the card reader is really simple, even though I did it wrong the first time, because I got confused by the labels on the converter. The 5 volt signal coming from the reader is connected to the RXI on the HV (high voltage) side of the converter, which makes a 3.3 volt signal available on the RXO pin on the LV (low voltage) side, which is then connected to the Rx pin of the RPi. I found several descriptions of how to connect this converter, but the clearest I found was actually an image on *hackaday.com* [hn.my/sparkfun].

Reading from serial

After this was connected correctly, reading RFID cards on the serial port can be done in only a couple of lines of code:

```
# import the serial library
# providing all functionality to
# interact with serial ports
# import serial

# "/dev/ttyAMA0" is the name of
# the serial port on the Raspberry
# Pi the RFID reader from
# Seeedstudio sends serial data at
# a baudrate of 9600 a timeout of
# 1 second will wait for data on
# the serial port for one second
# before continuing
port = serial.Serial("/dev/
ttyAMA0", baudrate=9600, time-
out=1)

while True:
    # the RFID reader sends the
    # data for one tag
    # as a 14 character string
    rcv = self.port.read(14)
    print rcv
```

Even though this will successfully display the raw data from the RFID tag, it's not actually the ID of the card. The details are available in the wiki of the reader [hn.my/rfidwiki], but since this product has been retired by Seeedstudio since I got it, there won't be much value in me explaining it. The only information on how to get the actual card ID I found was this C library [hn.my/rfidlib]. It was actually pretty trivial to express in Python code and can be found in the code that runs on the audio book player here. [hn.my/rfidpy]

Reading the RFID cards was easy, but as soon as I saw how the reader worked, I realized there was a flaw in my plan. I really wanted the play and pause of the audio playback to be controlled by the RFID card only. Placing the card on top would start playing

the corresponding book, and removing it would pause it. I assumed I could "ping" the RFID reader for the ID of the card within its range, but instead of this, the reader sends the ID of the card over serial as soon as it's in range. It does this only once. This meant I needed an additional button on the reader to be able to pause/resume playback. A small deception, but four buttons is still very acceptable.

3 Interrupts and Thread Safety

How it all starts

When the RPi that powers the audio book reader boots, it starts a service called supervisord [supervisord.org]. Supervisord is a process that can be configured to keep other processes running. If, for whatever reason, the code on the RPi crashes, supervisord will notice and restart it. An advantage of using supervisord is that it daemonizes my code, so I don't have to worry about daemonizing it myself (if you were to run the python code from the command line, it would stay in the foreground). Supervisord is also configured to start `main.py` [hn.my/mainpy] as soon as the RPi boots, making the reader ready to be used.

The main loop

Let's take a look at this file. If we execute `main.py`, it will create an instance of `BookReader` and call the `loop` method on it. Important to understand here are the following lines:

```
def loop(self):
    while True:
        rfid_card = self.rfid_
reader.read()

        if not rfid_card:
            continue
```

I left most of the code out, but the above lines show that this function enters in an endless loop. In each iteration of the loop, the `read` method is called on the `rfid_reader`object that is set on the book reader object. The most important lines in this method (leaving out some error handling) are:

```
def read(self):
    rcv = self.port.read(self.
string_length)

    if not rcv:
        return None

    tag = { "raw" : rcv,
     "mfr" : int(rcv[1:5], 16),
     "id" : int(rcv[5:11], 16),
     "chk" : int(rcv[11:13], 16)}

    return Card(tag)
```

We see that serial data is being read from `self.port`, which is an instance of `serial.Serial`. This port was setup with a timeout of one second, which means this line of code will block for a maximum of one second. If during that second serial data was received on the port, the `rcv` variable will contain that data which is then used to instantiate and return a `Card` object. If no data was returned, the `rcv` value will contain a `None` object. You may remember from the previous section that putting an RFID card on the reader only causes the ID to be sent once. This means that this `read` method will almost always block for precisely one second.

Interrupts

The simplest way to check if a button is pressed, is to keep checking the state of the button in a loop, and wait for it to change. However, if the main loop spends most of its time being blocked by waiting on data on the serial port, we can't really use this loop to see if my grandfather has pressed a button because a button press is usually a lot shorter than one second so we may miss it if the button is pressed and released within the second that the loop was blocked on the serial port.

This is where interrupts come into play. The main idea is that instead of continuously checking the state of button ourselves from the code, we can use the button to send a signal to the processor and only act if this happens. If we take a look at the `setup_gpio` method in the `BookReader` class, we see how this is setup for the buttons of the book reader. This method loops through the config values to setup each button. If we were to extract the setup of one of the buttons, it would look like this (I'm leaving out the last arguments on purpose, because they're not relevant just yet):

```
GPIO.setup(9, GPIO.IN)
GPIO.add_event_detect(9, GPIO.
FALLING, callback=self.player.
rewind)
```

In the first line, we're setting up pin 9 (which is one of the physical pins on the RPi board) to be an input. After that we're setting up this pin to listen to interrupts using the `add_event_detect` method. The second argument here (`GPIO.FALLING`) says we want to listen for an edge triggered interrupt, and more specifically the transition of the voltage on the pin from high to low (the falling edge). If this happens, we want to call the `rewind` method on the `player` object that is set on the book reader object. In short: if the voltage on pin 9 drops from high to low, we call `self.player.rewind`.

The other three buttons function in the exact same way. They're all connected to their own pin, and all have their own callback on the `self.player` object.

Threads and thread safety

The main loop described above runs in the main thread of the program. It keeps looping and blocking on the serial port. If an interrupt occurs on one of the button's pins, a separate thread is created to execute the code of the callback in parallel with the main thread. This means that the main loop is not blocking the new thread. If, for example, the pause button is pressed, a new thread is created, and the `pause` method is executed, which sends an instruction to the mpd server (that's playing the audio) to tell it to pause.

If you're using multiple threads like this, strange things can happen though. For example, within the main thread, the mpd server is constantly queried to get the current status. As you may remember, this information is transmitted over a local port. If I don't keep thread safety in mind, pressing the pause button will spawn a thread that also communicates over this local port, and the two information streams will interfere. I actually encountered this bug while developing this code, which manifested itself by occasionally throwing an exception when I pressed a button while a book was playing. The pause command to the mpd server (in the new thread) received information it shouldn't, and the status command in the main thread was receiving an incomplete one.

The problem is that both threads are sharing a resource (the mpd server), and they're doing it at the same time. A solution (the one that I chose), is for a thread to lock access to the resource while it's using it. I extended the `MPDClient` class into my own `LockableMPDClient`.

```
class
LockableMPDClient(MPDClient):
    def __init__(self, use_
unicode=False):
        super(LockableMPDClient,
self).__init__()
        self.use_unicode = use_
unicode
        self._lock = Lock()
    def acquire(self):
        self._lock.acquire()
    def release(self):
        self._lock.release()
    def __enter__(self):
        self.acquire()
    def __exit__(self, type,
value, traceback):
        self.release()
```

When instantiating the mpd client, we give the object a `threading.Lock` object, which provides a very easy locking interface. Since the mpd client object is shared by all threads, once one thread has acquired the lock, another one can't acquire it until it's released. If you're familiar with python, you'll notice the `__enter__` and `__exit__` methods. Providing these two methods allow me to do the following whenever I need to call a method on the mpd client:

```
def get_status(self):
    with self.mpd_client:
        return self.mpd_client.
status()
```

When a `with` statement is executed, the `__enter__` method is called on the object. When all code within the `with` block is executed, `__exit__` is called on the object, meaning that for the duration of `self.mpd_client.status()` access to the mpd client is locked for all other threads. Actually this use of `with` is only half of what it can do, because I don't need context guarding, but it is enough to achieve locking.

4 Finishing Up

The Buttons

The buttons may seem like the easiest part because a button is a very simple device. In the previous post, I told you about interrupts and how they make the buttons work: the program detects changes in the level on the pins to which the buttons are connected, and executes corresponding code in a separate thread.

Button Bounce

The problem, though, with any button is that they "bounce:"

> *Contact bounce (also called chatter) is a common problem with mechanical switches and relays. Switch and relay contacts are usually made of springy metals. When the contacts strike together, their momentum and elasticity act together to cause them to bounce apart one or more times before making steady contact. The result is a rapidly pulsed electric current instead of a clean transition from zero to full current. The effect is usually unimportant in power circuits, but causes problems in some analogue and logic circuits that respond fast enough to misinterpret the on off pulses as a data stream. From: Wikipedia*

If this problem isn't solved in hardware (by using a capacitor) or in software (by detecting quick changes and

waiting for the signal to settle), the program will interpret the bouncing of the button as multiple button presses and will behave in unpredictable ways.

You may remember the following code snippet from the previous section:

```
GPIO.setup(9, GPIO.IN)
GPIO.add_event_detect(9, GPIO.
FALLING, callback=self.player.
rewind)
```

If you compare it to the code I actually use, you'll see I pass an extra argument `bouncetime` to `GPIO.add_event_detect`, which is a value in milliseconds. I am not absolutely sure what the GPIO library does internally with this value, but I found the optimal values by experimentation, and by adjusting them later on when my grandfather was experiencing problems (I found out he keeps the buttons pressed a lot longer than me). Looking back, I think it would have been better to go for hardware debouncing, because the current version seems somewhat picky. My impression is that the complete sequence of press and "unpress" need to fall in the debounce time. But I could be wrong because I was adjusting these values over SSH from Canada while my brother was interpreting the results from what he observed when my grandfather used reader.

Pull Up vs. Pull Down

You may also remember from the previous post that the program detects the falling edge on the button pin to run the corresponding code. As a reminder: this means that it's waiting for the voltage on the pin to go from high (3.3 v) to low (0 v). This normally means we need to make sure that the voltage on the pin is pulled up to a default state of 3.3 volts. This is done by connecting the pin, through a very high value resistor, to the + 3.3 volts pin on the RPi. However, the RPi has built-in pull up and pull down resistors, so we don't need to do this. We can activate this with the `pull_up_down` argument to `GPIO.setup()`.

In the previous section I left out button bounce and pull up to describe how I use interrupts. Adding these two to the previous example, we get:

```
GPIO.setup(9, GPIO.IN, pull_up_
down=GPIO.PUD_UP)
GPIO.add_event_detect(9, GPIO.
FALLING,
   callback=self.player.rewind,
bouncetime=1000)
```

The button connects the pin to ground through a high value resistor (1.2 k ohms, if I remember correctly) when pressed to make the state of the pin low.

The Status Light

One detail I added, a bit for my own pleasure because I thought it looked nice, was a status light on the front of the reader. It has the following functions: it's off when the reader is powered off or booting. It's on when the player has booted and is ready to use. It blinks slowly while playing. It will give three fast flashes when a button is pressed or an RFID card is placed on the reader. It blinks once every 1.5 seconds if the player is paused.

The logic for the status light runs in a separate thread. A slightly simplified version looks like this:

```
# instantiate a status light
object, and tell it # the light is
connected to pin 23
status_light = StatusLight(23)

# start a new thread and give it
the start
# method on the status light
object as target
thread = Thread(target=status_
light.start)

# start the new thread
thread.start()
```

While the status light object is looping through "on" and "off" states in a pattern ("on" and "off" for blinking, just "on" for on, etc.), the main thread can set the `action` property on the status light object to the name of a different pattern. For example here [hn.my/playerpy70] the property is set to `blink`. The currently running pattern can also be interrupted by a different pattern by calling the `interrupt` method on the status light object. If, for example, the player is playing a book, and the light blinks slowly, a button press action can insert three quick flashes into the running pattern, after which the active one continues.

One year later

I spoke to my grandmother today because it's her birthday, and almost one year after having finished the reader, my grandfather still uses it daily, and proudly shows it to anyone who's visiting. He started requesting for music on it too, and whenever the reader is at my brother's house, he's having a hard time not being able to use it. I'm so happy that this little project was able to give some pleasure to a person that's been so enormously important in my life. It's harsh to say it with these words, but when I saw him last year, I was afraid he was close to being bored to death, literally. ■

Willem van der Jagt is a Dutch developer from Montreal and father of three. He has a passion for software design, and technology that helps people. He loves to learn new things. Willem is the lead developer at CakeMail.

Reprinted with permission of the original author.
First appeared in *hn.my/bookreader* (willemvanderjagt.com)

There Are No B Players

By DANIEL TENNER

"Only hire A players! Fire the B players!"

"If you hire B players, then they will hire C players!"

"Over time, the A players will get frustrated with the B players and will leave to go to other companies and you'll be left only with B and C players, unless you regularly cull B players."

HANDS UP IF you've read this advice before. Keep your hands up if you've believed it. I see that's all of you still. Now keep your hand up if you think you're a B player, that it's your nature to be one, that you'll never be an A player. Oh, where'd all the hands go?

"Everybody is a genius. But if you judge a fish by its ability to climb a tree, it will live its whole life believing that it is stupid."
— Albert Einstein

I was a B player

Once upon a time, I worked for Accenture. I started out fairly motivated and did some good work (or so it felt) in my first year, and then I got progressively more demotivated. I have no doubt that most of the people I worked with, or at least most of the people who had to rate my performance, rated me as a B player. Not a bad contributor, but not the kind of balls-to-the-walls excellence that they hoped for from a super-keen, motivated Accenture consultant.

I missed two rounds of promotions before I finally left Accenture. In theory this was due to one-off structural stuff happening while I was there (like Accenture taking a $450 million write off on the NHS project), but I knew that if I had been rated as one of the top people, they would have figured out a way to promote me even during a promotion freeze (Accenture works like that, with special deals for special people). So I was clearly not at

the top. At Accenture, I was a consistent B player.

Even in my two subsequent start-ups I was a B player. It turns out that I don't operate at my full potential when I believe someone else will find and fix my mistakes. I play better without a safety net. I also have a burning need to work on stuff that I feel I own completely. The two combined mean that on Vocalix and Woobius, I was working at maybe 10–20% of my capacity at the time (probably less than 5% of my current capacity). I was not in a state of flow. I was easily distracted. I frequently felt demotivated because of what I perceived as unfair ownership/shares split. I still got stuff done, of course, but most people who are not in an absolutely abysmal environment will get shit done.

Long before these events, I was a B player at school and then at university. I might have been smart, but I never felt like putting in the seemingly unending amounts of largely pointless effort that academic excellence would have required. Add to this that I was undisciplined, didn't have many friends, and in fact was constantly bullied in my early years of school. For most of school, I was a B player, if not a C player.

So, shall we consign me to the B player trash can and forget about this person called Daniel Tenner? Or, as my dad suggested, in a skilful reductio ad absurdum, to the headmaster who declared me "unsalvageable" and wanted to expel me, "so do we take him out back and shoot him now?"

People are not cogs

The A/B player mentality comes out of a worldview where people are replaceable cogs in a machine that you're building to make money. In this context, they are measured mostly by their ability to produce a positive effect on the bottom line. Sure, there may be some qualities or defects that don't have an immediately apparent effect on profits, but in this worldview, it all comes down to the numbers in the end, to one number in particular: profit.

Within that worldview, the concept of A and B players makes sense. An A player has an outsized positive effect on your profits. A B player has a more moderate positive effect. A C player may have no effect or worse. It stands to reason that the best thing to do in this context is to have only A players: this way you'll have more revenue, more opportunities being grabbed, and fewer people to share the pot with. If that's all that matters to you, then please disregard my article: it's not addressed to you.

If, however, the thought of measuring your entire human output with a single number makes you shudder or at least makes you a little bit uncomfortable, please read on.

Human beings are deep, complex creatures with many subtleties and nuances. They can contribute to a variety of endeavors in a whole lot of ways. The key to unlocking this human potential in yourself is to find the stuff you're good at, that you enjoy doing, and that you think is worth doing to make a positive difference. Then find that elusive state of flow where work becomes more like play, where despite dealing with a variety of tasks (some of which may seem boring), you take the time to love what you do and do what you love. When you find that place, you're an A player.

Everyone seems perfectly willing to accept the above statement when it comes to their own self. Even better, we all breathlessly repeat this pearl of wisdom to friends, family, and sometimes complete strangers that we feel some sympathy towards. We believe in its deeper Truth, on its positive impact on our lives.

And yet when it comes to hiring and firing, we suddenly conclude that some people are hopeless B players to be culled, lest they pollute our precious company by hiring even worse examples of themselves, or setting a low hiring standard for the whole company.

That is elitist crap, merely there as a consequence of a narrow-minded worldview and as an escape hatch to allow us to blame poor performance on other people rather than ourselves. There are no B players, only people whose potential is not being brought to life, fish which are made to climb trees and then told they suck.

A better view of hiring

Pretty much everyone in the world has the potential to make a great contribution to some human endeavor. Sure, some people are cleverer or stronger or faster or more nimble or more diligent or more patient or more helpful or more of a zillion different qualities humans can be evaluated on. However, this contribution is only possible when the said human being is placed in a context that gets the best out of them.

Most of humanity labors in terrible, dehumanizing, boring, uninspiring contexts. Too many still are slaves, or toil for survival or safety or comfort rather than for inspiration, fulfilment, or any kind of meaningful purpose. That is a tragedy first of all for ourselves as a species, as we miss out on great contributions from billions of people who could give so much more to the world around them. A few of us are lucky to be able to find or fashion an environment which enables us to give our best day after day. Calling the latter "A players" and firing the rest is not only

callous, it is immensely short-sighted and bone-headed on both a personal, a business, and a societal level.

When it comes to hiring, not everyone is right for your company. Some people will thrive in the open environment we've built at GrantTree. Others will excel in a numbers-and-measurements-oriented, strictly hierarchical company. Others yet will thrive in a socially oriented context where they feel part of a family. Others may give their best when surrounded by chaos and relying on themselves alone. To make matters worse, people will shift between these and other categories throughout their life, depending on many factors including personal growth, external demands on their resources, etc. In addition, people have skills, abilities and aspirations that will determine whether there is useful work for them to do within a given company.

Anyone who thrives in the environment you've built for your company and wants to contribute something important will be by definition an A player. Your job when recruiting is to find those people who will do well in the environment you've built, and who have skills, abilities, and aspirations that complement the needs of your company. Depending on how different your company is from the norm, there may be no one who fits so well outright. Perhaps they will need some coaching to embrace your unique culture. Perhaps some training to learn the ropes. Your job then becomes to find people with the right potential to thrive in your company, and then to coach them, train them, and help them to unfold their potential.

Whatever you do, though, don't make the mistake of thinking that those who don't fit your specific environment are unworthy human beings, categorized forever with the "B" brush stroke, unlikely ever to amount to much, and don't let yourself fall into the trap of thinking you're better than them. You're not better, you're merely in a better place, and with some humility perhaps you will be able to see that your role is not to sort the deserving from the unworthy, but merely to help those whose way you're lucky to cross to contribute at their best, whether in your company or somewhere else. ■

Daniel Tenner is the founder of Woobius and GrantTree. Known as "swombat" on Hacker News and Twitter, he is now producing *swombat.com*, a daily updated resource for people who like to read startup articles.

Reprinted with permission of the original author.
First appeared in *hn.my/bplayer* (*danieltenner*.com)

What Writing — And Selling — Software Was Like In The '80s

By ANDREW WULF

I STARTED MY CAREER in 1981, working for 3 years at a defense contractor. By 1985 I started my first company to develop and then sell a spreadsheet-like application for the Mac called Trapeze. It shipped in January 1987, but by the end of the year we sold it, and then I started a new company to just develop for other people. We worked on the presentation app Persuasion (for the author) and then spent 6 years working on Deltagraph for its publisher.

So what was it like back then in the dark ages? Quite different from today in many ways; not so different in others. Warning, antique history!

I quit my job in late 1984 and then came up with the idea that became Trapeze. Like any young person with an idea, I got people I knew excited and we got a group of investors together. As it is today, there were people who liked to invest in new ideas, but unlike today many of them had no idea what we were doing. The whole idea of software was unfamiliar to many. I remember talking with a banker who upon hearing we were going to work on software thought we were making lingerie!

There was little email (at my first job I had an email address outside of work but I only knew one person with one — my boss — and we sat next to each other at work) and of course, there was no internet. Learning meant libraries, or magazines, or maybe a user group. If you wanted to know what a piece of software did, you had to buy a copy.

Just finding out what software existed meant reading ads or magazine reviews or attending a computer show.

We started development in late 1985. We had two 512K Macs and one Mac XL (a Lisa running MacOS) that had a small hard drive we all shared. I used the XL, and the other two developers could access the hard drive over Appletalk. At first we used some C compiler whose name I don't remember. It was pretty slow. The linker spent most of its time drawing icons on the screen. At some point we started using Lightspeed-C (later named Think C), which helped a lot. Even though Apple had mostly Pascal interfaces, we used C, because I thought C was the future.

Development was slow as we were basically inventing a new idea, working in a new language on a new platform, and on ridiculously slow machines (5 and 8 MHz) with tiny screens compared to today. In May of 1986 I went to the first Apple developer conference (not yet named WWDC) where basically the entire world of Mac developers showed up — we all fit in a single hotel ballroom! During the week, Apple took us out on a boat in SF harbor for some fun. We all thought it would be funny if we sank and the whole Mac industry vanished with us.

Today people think everyone did Waterfall in the old days. We didn't even know that word, and we never gave much thought to processes. We organized the app development into three pieces with a reasonable contract on an informal API. In fact it had to be informal as when we started C didn't even have prototypes — you had to manually make sure a function's parameter and calls matched! We also had no repository as they weren't available on MacOS at the time, so we had to have a manual process to keep track of files. I did all the "official" builds.

In August we all went to Macworld Boston where we started doing press demos in a hotel room as well as wandering the floor looking at other people's apps. This was the first time I really saw what other people were doing (remember no websites, no free demos, spending money to even see an app) and was horrified by the interface I had designed. It reeked. Now that I could see other people's work it didn't measure up.

I would work 90-hour weeks for the next four months to completely rewrite the interface while supporting the old one so the other two could keep working. Of course while I was doing this I had to talk with the press, do demos, deal with investors, find suppliers, hire people, and all the usual business stuff. Unlike today you had to do everything yourself. Jolt Cola was my friend.

We finally shipped it at Macworld SF in January 1987.

Now what does that mean? Today shipping is nothing, push a few buttons and it's uploaded somewhere. In those days shipping meant floppy disk duplicators, printers for manuals, boxes, and actual shipping. Who did you ship to? Distributors and mail order houses. You rarely sold to end users. Distributors took cases of boxes, putting a short description into a paper catalog they gave to retailers. If they sold any they sent you a check 90-180 days later. Anything they didn't sell came back 6 months later. Mail order usually paid quicker. Distributors would pay you around 30% of the retail price; the mail order people were a little better. If you wanted a retailer to stock your app you were expected to advertise; no one did anything free for you other than put you in a catalog. This made making money a pain in the ass.

Of course potential customers had to figure out you existed, demand you from their retailer who hopefully ordered from the distributor. If they did buy a copy you only found out who they were if they filled out a registration card or called for support. When I think back at how crappy this all was I wonder why I ever got into it! Today it all sounds stupid.

We got a good review in Macworld, but the guy who wrote the MacUser review had a bad day and the review was horrible. Of course these were written in January and only came out 3 months later. The one bad review killed our sales. When the only source of information is reviews it only took one bad one. Being a small developer we couldn't fix it fast enough — it took months to make changes, ship it, and then wait for an updated review 4 months or so later in the magazine. A year later we met the author and he admitted he hadn't been fair and took his personal issues out on us. We sold Trapeze to a company in Boston which then split and formed Deltapoint in California. Eventually we would start Deltagraph for them.

Apple helped bring my second company together with the author of a drawing program who wanted to make a presentation program out of it. He had seen Cricket Presents in an early alpha and thought he had the right stuff to do one as well, but knew nothing of charts, so that's where we came in. He and I sat in his condo in Brooklyn for three days and figured out what a presentation package should look like. He had briefly seen Presents, and I knew nothing, so we just made it up.

We worked in Texas and he was in New York, so since there was no email, we worked out a system by sending first floppies and later cartridge drives back and forth every other day, merging each side's changes by hand. Persuasion shipped in August, 1988. He had it published by Aldus. Eventually in the '90s it was acquired by Adobe

who killed it as PowerPoint become a virtual monopoly by being part of Office.

So in late 1988, we and Deltapoint decided to start building a charting and graphing program to challenge the market leader, Cricket Graph. Of course it was on Mac; Windows was not a viable platform until 3.1 in 1992. Everything appeared first on the Mac.

With Deltagraph we again wrote in C. We had four programmers including me, plus one QA. The publisher had a product manager and another QA, plus a lot of support people. They were in California and we were still in Texas. Until 1990 we had no reliable email that could send binaries, so we still used FedEx. Until the last version we did in 1993, we were the only programmers. Each version would start with a page or two of ideas. We again broke the development into pieces with careful APIs. We still didn't have any repository software. Now that we had a real product manager, we found it worked best to talk on the phone about an idea first; then often I would prototype it in HyperCard, Apple's nifty little app. Usually that resulted in speculative coding. We would write enough to build a version and send it via FedEx so that the product manager could see it. This would go back and forth until it was happy, or tabled. Note there was no advance planning; this was real lean type development long before agile was a thing. Everything in Deltagraph was built in parallel streams during the usual six-to-eight month development cycle.

Now I had read at some point the famous Byte magazine Smalltalk issue and wanted to use OO programming in Deltagraph. Of course there was no language I could use yet, so I rolled my own extensions to C, some incredibly lame ones involving switch statements. It made it easy to have a single output driver, and "subclasses" for each output type. This became Deltagraph's biggest feature; it produced Postscript and Adobe Illustrator native format files which meant you could build a complex chart in Deltagraph and then have your artist monkey with it in Illustrator. I reverse engineered their format. This format later become the basis of PDF.

We barely finished Deltagraph before the publisher ran out of money. We were actually owed $150,000 by the shipping date, but it was a big hit and became a huge money maker for them. We worked on five major release.

In those days, you almost never sent out patch disks. Generally you had to wait six-to-eight months to ship a new version and you had to charge the customer for the update. Trapeze only needed one floppy but Deltagraph shipped on something like 10 disks. Paying for hundreds of thousands of disks is expensive; add to that printing of manual updates and

boxes and shipping and you never did this casually. So the version we sent to the duplicators had to be perfect and live for months. I was always the final arbiter of what shipped. Thankfully we never had an issue with either four versions of Trapeze or the five of Deltagraph.

Of course we didn't write unit tests or anything like that. I never even heard the term until ten years later. But we tested the builds continuously every day and kept careful record of anything that didn't work correctly. Having QA use the app all day every day from start to finish meant it was well tested by the ship date. It also helped in finding irritations in features or UI before the customers found it — use an app for months and everything bad is magnified! I still believe strongly that hard core continuous QA produces quality apps. I still try to get people to do this today, and it gets the same results.

Deltapoint eventually sold Deltagraph in the mid '90s, and it wandered around but is still available today — sadly still from the same codebase we started in 1988! It has to be seriously awful today. We wanted to rewrite it in C++ in 1993 but Deltapoint said no. Now twenty years and hundreds of engineers later, it must be horrible. But for the longest time it was the standard for printed charts.

Hopefully you get a vague idea of what writing — and selling — software was like back then. Everything is so much easier today, but users expect so much more. You need to ship continuously; feedback is instant, but so is disgust; and you have to know way more technologies to write anything. Back then all I needed was K&R C, Inside Macintosh and imagination.

The only thing I miss is the constant opportunity for invention: when you are doing something brand new and have nothing to help you, it's all up to your imagination and creativity. There was no internet, no Google, no StackOverflow. It was just you and your friends and your brain.

Other than that I don't miss it at all, but it was fun! ■

In 3 decades of programming Andrew has worked on almost every kind of software. Currently he works in mobile at a well known travel brand and writes in his blog, *thecodist.com*

Reprinted with permission of the original author.
First appeared in *hn.my/software80* (thecodist.com)

How to Read a Patent

By ERIC ADLER

Patents are complex documents that bury a handful of important sentences under a mountain of fluff and jargon. If you're going to read a patent (and I urge you not to) you might as well start with the important parts, and read them correctly.

Let's suppose you want to figure out whether your new technology might infringe some patent. Here's a simple strategy I might use to start the infringement analysis.

First, skip down to the "claims."

The claims are a numbered list of run-on sentences buried toward the end of the patent. Although they come last, the claims are actually the meat of the patent. They define the actual patent rights. The other sections are auxiliary; they are supposed to help explain the claims.

Next, highlight all the **"independent claims."** These are the claims starting with the word "A". For example, "1. A swizzle stick adapted to…" or "9. A computer implemented method for…." There are probably 2 or 3 of these independent claims, but there can be more.

The others claims are the **"dependent claims."** The dependent claims start with a phrase like "The ____ **of claim** _____" and refer back to a previous claim. For example, "7. The swizzle stick of claim 1, further comprising…" Ignore the dependent claims for now. Google Patents actually displays dependent claims in gray text, making it easy to skim past them (UX!).

In 2 minutes, we've narrowed a huge patent document down to a small handful of important sentences: the independent claims. Next we will break down the independent claims and compare them to our technology.

Any Infringement?

We want to determine whether our technology would infringe this patent we're reading. It will infringe if it incorporates *every element* of any one of the claims. Fortunately, we only need to check the independent claims at this point.

Claims read like run-on sentences, but if you're lucky the run-ons will be broken down into sections and even subsections. Take a look at the first claim in our annotated patent. The first claim is a method with 3 steps, and the first step has 4 qualifications:

1. A computer implemented method of scoring a plurality of linked documents, comprising:

- Obtaining a plurality of documents,
  - At least some of the documents being linked documents,
  - At least some of the documents being linking documents, and
  - At least some of the documents being both linked documents and linking documents,
  - Each of the linked documents being pointed to by a link in one or more of the linking documents;
- Assigning a score to each of the linked documents based on scores of the one or more linking documents and
- Processing the linked documents according to their scores.

If your technology does not incorporate even one of these steps or qualifications, it's (probably) not infringing this claim. Let's say your technology does 90% of the things described in claim 1, except that none of the documents are both "linked documents and linking documents" (as required by the claim). Then your technology is (probably) not infringing on this claim 1. It's an all-or-nothing analysis, at least at this preliminary stage. If even one element of the patent claim is missing from your technology, your tech is (probably) not infringing on the patent.

Parts of the claim might be ambiguous on their own. So at this point, we start referring back to the rest of the patent document to try to understand whether words in the claim have some special meaning. This type of claim interpretation analysis is complicated, and beyond the scope of this post. In theory, if you are a reasonably competent engineer/scientist in the field of this patent, the claims should be written in language you can understand. (This is rarely true, but in theory, it's required).

Annotated Patent

When reading a patent, first skip to the independent claims and read them carefully. The rest of the document is far less important. Here's an annotated patent explaining some of the other parts of the patent document.

How To Read a Patent

The section called the "*Claims*" defines the actual patent rights. I'll point the claims out in red. The rest of the patent document is supposed to help us understand and interpret the claims.

- Not Patent Rights
- Patent Rights

Strategy for Reading a Patent

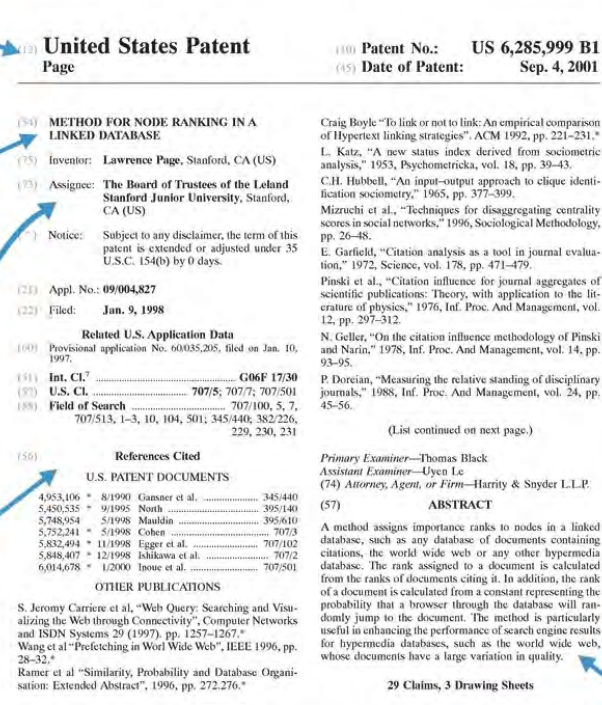
1. Start by reading the first "claim." It will be buried towards the end of the patent, but its the most important part.
2. Try to put the claim in context by skimming through the drawings and reading the "summary" section. Hopefully the first claim will start to make some sense.
3. Someone is **infringing** the patent if their technology incorporates every element of one of the patent's claims. [Your milage may vary. Patent infringement rules are extremely complicated.]

INVENTOR - The patent must correctly identify the inventor(s). However, the "assignee" owns the real patent rights.

TITLE - This is just a name. The title might suggest a broad patent, but only the claims define the actual patent rights.
Don't be fooled by broad-sounding titles. In fact, just ignore the title.

ASSIGNEE - The true owner. This assignee is often the company that the inventor works for.

REFERENCES - some documents the examiner reviewed before granting this patent. More references may suggest a stronger patent.



(12) **United States Patent**
Page

(10) **Patent No.: US 6,285,999 B1**
(45) **Date of Patent: Sep. 4, 2001**

(54) **METHOD FOR NODE RANKING IN A LINKED DATABASE**

(75) Inventor: **Lawrence Page**, Stanford, CA (US)

(73) Assignee: **The Board of Trustees of the Leland Stanford Junior University**, Stanford, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/004,827**

(22) Filed: **Jan. 9, 1998**

Related U.S. Application Data

(60) Provisional application No. 60/035,205, filed on Jan. 10, 1997.

(51) **Int. Cl.**[7] **G06F 17/30**
(52) **U.S. Cl.** **707/5**; 707/7; 707/501
(58) **Field of Search** 707/100, 5, 7, 707/513, 1–3, 10, 104, 501; 345/440; 382/226, 229, 230, 231

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,953,106	*	8/1990	Gansner et al. 345/440
5,450,535	*	9/1995	North 395/140
5,748,954		5/1998	Mauldin 395/610
5,752,241	*	5/1998	Cohen 707/3
5,832,494	*	11/1998	Egger et al. 707/102
5,848,407	*	12/1998	Ishikawa et al. 707/2
6,014,678	*	1/2000	Inoue et al. 707/501

OTHER PUBLICATIONS

S. Jeromy Carriere et al, "Web Query: Searching and Visualizing the Web through Connectivity", Computer Networks and ISDN Systems 29 (1997). pp. 1257–1267.*
Wang et al "Prefetching in Worl Wide Web", IEEE 1996, pp. 28–32.*
Ramer et al "Similarity, Probability and Database Organisation: Extended Abstract", 1996, pp. 272.276.*
Craig Boyle "To link or not to link: An empirical comparison of Hypertext linking strategies". ACM 1992, pp. 221–231.*
L. Katz, "A new status index derived from sociometric analysis," 1953, Psychometricka, vol. 18, pp. 39–43.
C.H. Hubbell, "An input–output approach to clique identification sociometry," 1965, pp. 377–399.
Mizruchi et al., "Techniques for disaggregating centrality scores in social networks," 1996, Sociological Methodology, pp. 26–48.
E. Garfield, "Citation analysis as a tool in journal evaluation," 1972, Science, vol. 178, pp. 471–479.
Pinski et al., "Citation influence for journal aggregates of scientific publications: Theory, with application to the literature of physics," 1976, Inf. Proc. And Management, vol. 12, pp. 297–312.
N. Geller, "On the citation influence methodology of Pinski and Narin," 1978, Inf. Proc. And Management, vol. 14, pp. 93–95.
P. Doreian, "Measuring the relative standing of disciplinary journals," 1988, Inf. Proc. And Management, vol. 24, pp. 45–56.

(List continued on next page.)

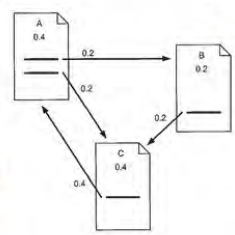
Primary Examiner—Thomas Black
Assistant Examiner—Uyen Le
(74) *Attorney, Agent, or Firm*—Harrity & Snyder L.L.P.

(57) **ABSTRACT**

A method assigns importance ranks to nodes in a linked database, such as any database of documents containing citations, the world wide web or any other hypermedia database. The rank assigned to a document is calculated from the ranks of documents citing it. In addition, the rank of a document is calculated from a constant representing the probability that a browser through the database will randomly jump to the document. The method is particularly useful in enhancing the performance of search engine results for hypermedia databases, such as the world wide web, whose documents have a large variation in quality.

29 Claims, 3 Drawing Sheets

PRETTY PICTURE - this is literally just a picture slapped onto the cover to look cool.



ABSTRACT - A quick summary of the technology involved in the patent. The abstract may suggest a broad or improbable technology, but only the claims define the actual patent rights.

DRAWINGS

The drawings help explain or interpret the claims. But only the claims define the patent rights.

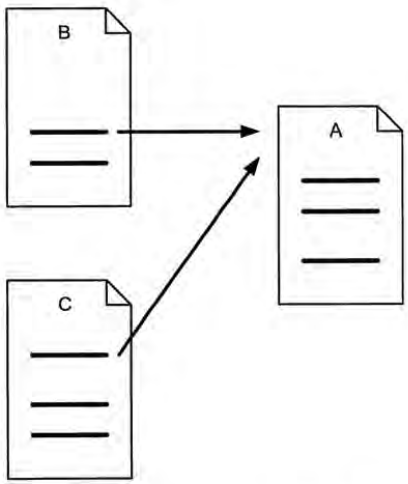


FIG. 1

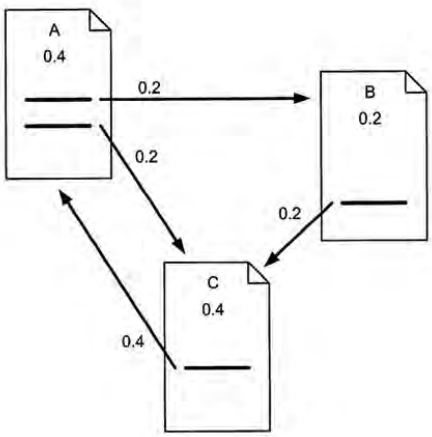


FIG. 2

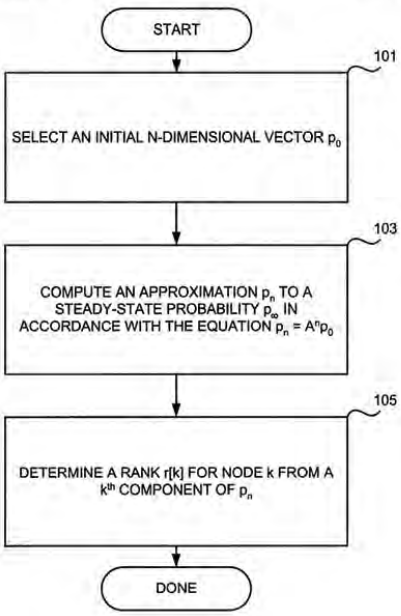


FIG. 3

METHOD FOR NODE RANKING IN A LINKED DATABASE

CROSS-REFERENCES TO RELATED APPLICATIONS

This application claims priority from U.S. provisional patent application Ser. No. 60/035,205 filed Jan. 10, 1997, which is incorporated herein by reference.

STATEMENT REGARDING GOVERNMENT SUPPORT

This invention was supported in part by the National Science Foundation grant number IRI-9411306-4. The Government has certain rights in the invention.

FIELD OF THE INVENTION

This invention relates generally to techniques for analyzing linked databases. More particularly, it relates to methods for assigning ranks to nodes in a linked database, such as any database of documents containing citations, the world wide web or any other hypermedia database.

BACKGROUND OF THE INVENTION

Due to the developments in computer technology and its increase in popularity, large numbers of people have recently started to frequently search huge databases. For example, internet search engines are frequently used to search the entire world wide web. Currently, a popular search engine might execute over 30 million searches per day of the indexable part of the web, which has a size in excess of 500 Gigabytes. Information retrieval systems are traditionally judged by their precision and recall. What is often neglected, however, is the quality of the results produced by these search engines. Large databases of documents such as the web contain many low quality documents. As a result, searches typically return hundreds of irrelevant or unwanted documents which camouflage the few relevant ones. In order to improve the selectivity of the results, common techniques allow the user to constrain the scope of the search to a specified subset of the database, or to provide additional search terms. These techniques are most effective in cases where the database is homogeneous and already classified into subsets, or in cases where the user is searching for well known and specific information. In other cases, however, these techniques are often not effective because each constraint introduced by the user increases the chances that the desired information will be inadvertently eliminated from the search results.

Search engines presently use various techniques that attempt to present more relevant documents. Typically, documents are ranked according to variations of a standard vector space model. These variations could include (a) how recently the document was updated, and/or (b) how close the search terms are to the beginning of the document. Although this strategy provides search results that are better than with no ranking at all, the results still have relatively low quality. Moreover, when searching the highly competitive web, this measure of relevancy is vulnerable to "spamming" techniques that authors can use to artificially inflate their document's relevance in order to draw attention to it or its advertisements. For this reason search results often contain commercial appeals that should not be considered a match to the query. Although search engines are designed to avoid such ruses, poorly conceived mechanisms can result in disappointing failures to retrieve desired information.

Hyperlink Search Engine, developed by IDD Information Services, (http://rankdex.gari.com/) uses backlink information (i.e., information from pages that contain links to the current page) to assist in identifying relevant web documents. Rather than using the content of a document to determine relevance, the technique uses the anchor text of links to the document to characterize the relevance of a document. The idea of associating anchor text with the page the text points to was first implemented in the World Wide Web Worm (Oliver A. McBryan, GENVL and WWWW: Tools for Taming the Web, First International Conference on the World Wide Web, CERN, Geneva, May 25–27, 1994). The Hyperlink Search Engine has applied this idea to assist in determining document relevance in a search. In particular, search query terms are compared to a collection of anchor text descriptions that point to the page, rather than to a keyword index of the page content. A rank is then assigned to a document based on the degree to which the search terms match the anchor descriptions in its backlink documents.

The well known idea of citation counting is a simple method for determining the importance of a document by counting its number of citations, or backlinks. The citation rank r(A) of a document which has n backlink pages is simply

$$r(A)=n.$$

In the case of databases whose content is of relatively uniform quality and importance it is valid to assume that a highly cited document should be of greater interest than a document with only one or two citations. Many databases, however, have extreme variations in the quality and importance of documents. In these cases, citation ranking is overly simplistic. For example, citation ranking will give the same rank to a document that is cited once on an obscure page as to a similar document that is cited once on a well-known and highly respected page.

SUMMARY

Various aspects of the present invention provide systems and methods for ranking documents in a linked database. One aspect provides an objective ranking based on the relationship between documents. Another aspect of the invention is directed to a technique for ranking documents within a database whose content has a large variation in quality and importance. Another aspect of the present invention is to provide a document ranking method that is scalable and can be applied to extremely large databases such as the world wide web. Additional aspects of the invention will become apparent in view of the following description and associated figures.

One aspect of the present invention is directed to taking advantage of the linked structure of a database to assign a rank to each document in the database, where the document rank is a measure of the importance of a document. Rather than determining relevance only from the intrinsic content of a document, or from the anchor text of backlinks to the document, a method consistent with the invention determines importance from the extrinsic relationships between documents. Intuitively, a document should be important (regardless of its content) if it is highly cited by other documents. Not all citations, however, are necessarily of equal significance. A citation from an important document is more important than a citation from a relatively unimportant document. Thus, the importance of a page, and hence the rank assigned to it, should depend not just on the number of citations it has, but on the importance of the citing documents as well. This implies a recursive definition of rank: the

RELATED APPLICATIONS
One application can often branch off into several patents. This section keeps track of the prior applications in the chain.

FIELD OF INVENTION
Described the field of technology in general terms. But remember that only the claims define the patent rights.

BACKGROUND
Discusses the problems that this invention purports to solve. Since its not required, many patents include only a cursory "background" section.

SUMMARY
is supposed to quickly describe how the invention works. It does not define the patent rights. Only the claims grant patent rights.

to one of a few nodes that have a high importance, and will not take the surfer to any of the other nodes. This can be very effective in preventing deceptively tagged documents from receiving artificially inflated relevance. Alternatively, the random linking probability could be distributed so that random jumps do not happen from high importance nodes, and only happen from other nodes. This distribution would model a surfer who is more likely to make random jumps from unimportant sites and follow forward links from important sites. A modification to avoid drawing unwarranted attention to pages with artificially inflated relevance is to ignore local links between documents and only consider links between separate domains. Because the links from other sites to the document are not directly under the control of a typical web site designer, it is then difficult for the designer to artificially inflate the ranking. A simpler approach is to weight links from pages contained on the same web server less than links from other servers. Also, in addition to servers, internet domains and any general measure of the distance between links could be used to determine such a weighting.

Additional modifications can further improve the performance of this method.Rank can be increased for documents whose backlinks are maintained by different institutions and authors in various geographic locations. Or it can be increased if links come from unusually important web locations such as the root page of a domain.

Links can also be weighted by their relative importance within a document. For example, highly visible links that are near the top of a document can be given more weight. Also, links that are in large fonts or emphasized in other ways can be given more weight. In this way, the model better approximates human usage and authors' intentions. In many cases it is appropriate to assign higher value to links coming from pages that have been modified recently since such information is less likely to be obsolete.

Various implementations of the invention have the advantage that the convergence is very fast (a few hours using current processors) and it is much less expensive than building a full-text index. This speed allows the ranking to be customized or personalized for specific users. For example, a user's home page and/or bookmarks can be given a large initial importance, and/or a high probability of a random jump returning to it. This high rating essentially indicates to the system that the person's homepage and/or bookmarks does indeed contain subjects of importance that should be highly ranked. This procedure essentially trains the system to recognize pages related to the person's interests. The present method of determining the rank of a document can also be used to enhance the display of documents. In particular, each link in a document can be annotated with an icon, text, or other indicator of the rank of the document that each link points to. Anyone viewing the document can then easily see the relative importance of various links in the document.

The present method of ranking documents in a database can also be useful for estimating the amount of attention any document receives on the web since it models human behavior when surfing the web. Estimating the importance of each backlink to a page can be useful for many purposes including site design, business arrangements with the backlinkers, and marketing. The effect of potential changes to the hypertext structure can be evaluated by adding them to the link structure and recomputing the ranking.

Real usage data, when available, can be used as a starting point for the model and as the distribution for the alpha factor.

This can allow this ranking model to fill holes in the usage data, and provide a more accurate or comprehensive picture.

Thus, although this method of ranking does not necessarily match the actual traffic, it nevertheless measures the degree of exposure a document has throughout the web.

Another important application and embodiment of the present invention is directed to enhancing the quality of results from web search engines. In this application of the present invention, a ranking method according to the invention is integrated into a web search engine to produce results far superior to existing methods in quality and performance. A search engine employing a ranking method of the present invention provides automation while producing results comparable to a human maintained categorized system. In this approach, a web crawler explores the web and creates an index of the web content, as well as a directed graph of nodes corresponding to the structure of hyperlinks. The nodes of the graph (i.e. pages of the web) are then ranked according to importance as described above in connection with various exemplary embodiments of the present invention.

The search engine is used to locate documents that match the specified search criteria, either by searching full text, or by searching titles only. In addition, the search can include the anchor text associated with backlinks to the page. This approach has several advantages in this context. First, anchors often provide more accurate descriptions of web pages than the pages themselves. Second, anchors may exist for images, programs, and other objects that cannot be indexed by a text-based search engine. This also makes it possible to return web pages which have not actually been crawled. In addition, the engine can compare the search terms with a list of its backlink document titles. Thus, even though the text of the document itself may not match the search terms, if the document is cited by documents whose titles or backlink anchor text match the search terms, the document will be considered a match. In addition to or instead of the anchor text, the text in the immediate vicinity of the backlink anchor text can also be compared to the search terms in order to improve the search.

Once a set of documents is identified that match the search terms, the list of documents is then sorted with high ranking documents first and low ranking documents last. The ranking in this case is a function which combines all of the above factors such as the objective ranking and textual matching. If desired, the results can be grouped by category or site as well.

It will be clear to one skilled in the art that the above embodiments may be altered in many ways without departing from the scope of the invention. Accordingly, the scope of the invention should be determined by the following claims and their legal equivalents.

What is claimed is:

1. A computer implemented method of scoring a plurality of linked documents, comprising:

obtaining a plurality of documents, at least some of the documents being linked documents, at least some of the documents being linking documents, and at least some of the documents being both linked documents and linking documents, each of the linked documents being pointed to by a link in one or more of the linking documents;

assigning a score to each of the linked documents based on scores of the one or more linking documents and

processing the linked documents according to their scores.

DETAILED DESCRIPTION

The detailed description section may continue for several pages.

CLAIMS!!!

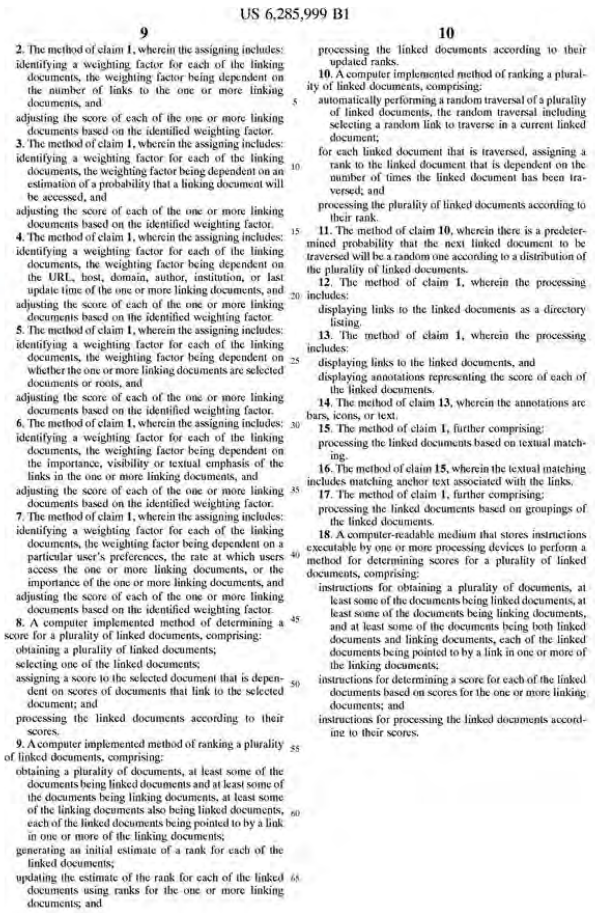
These are the weird run-on sentences that actually define the scope of the patent rights.

Independent Claims

Pay attention to independent claims. They start with the word "A" as in "A widget that..." or "A computer implemented method of..."

The independent claims are the broadest.

Dependent Claims
These claims branch off an independent claim and add additional limitations to the parent claim. They start with the word "The" as in "the widget of claim 1..."
A dependent claim is always more narrow than its parent independent claim.



US 6,285,999 B1

9

2. The method of claim 1, wherein the assigning includes:
identifying a weighting factor for each of the linking documents, the weighting factor being dependent on the number of links to the one or more linking documents, and
adjusting the score of each of the one or more linking documents based on the identified weighting factor.
3. The method of claim 1, wherein the assigning includes:
identifying a weighting factor for each of the linking documents, the weighting factor being dependent on an estimation of a probability that a linking document will be accessed, and
adjusting the score of each of the one or more linking documents based on the identified weighting factor.
4. The method of claim 1, wherein the assigning includes:
identifying a weighting factor for each of the linking documents, the weighting factor being dependent on the URL, host, domain, author, institution, or last update time of the one or more linking documents, and
adjusting the score of each of the one or more linking documents based on the identified weighting factor.
5. The method of claim 1, wherein the assigning includes:
identifying a weighting factor for each of the linking documents, the weighting factor being dependent on whether the one or more linking documents are selected documents or roots, and
adjusting the score of each of the one or more linking documents based on the identified weighting factor.
6. The method of claim 1, wherein the assigning includes:
identifying a weighting factor for each of the linking documents, the weighting factor being dependent on the importance, visibility or textual emphasis of the links in the one or more linking documents, and
adjusting the score of each of the one or more linking documents based on the identified weighting factor.
7. The method of claim 1, wherein the assigning includes:
identifying a weighting factor for each of the linking documents, the weighting factor being dependent on a particular user's preferences, the rate at which users access the one or more linking documents, or the importance of the one or more linking documents, and
adjusting the score of each of the one or more linking documents based on the identified weighting factor.
8. A computer implemented method of determining a score for a plurality of linked documents, comprising:
obtaining a plurality of linked documents;
selecting one of the linked documents;
assigning a score to the selected document that is dependent on scores of documents that link to the selected document; and
processing the linked documents according to their scores.
9. A computer implemented method of ranking a plurality of linked documents, comprising:
obtaining a plurality of documents, at least some of the documents being linked documents and at least some of the documents being linking documents, at least some of the linking documents also being linked documents, each of the linked documents being pointed to by a link in one or more of the linking documents;
generating an initial estimate of a rank for each of the linked documents;
updating the estimate of the rank for each of the linked documents using ranks for the one or more linking documents; and

10

processing the linked documents according to their updated ranks.
10. A computer implemented method of ranking a plurality of linked documents, comprising:
automatically performing a random traversal of a plurality of linked documents, the random traversal including selecting a random link to traverse in a current linked document;
for each linked document that is traversed, assigning a rank to the linked document that is dependent on the number of times the linked document has been traversed; and
processing the plurality of linked documents according to their rank.
11. The method of claim 10, wherein there is a predetermined probability that the next linked document to be traversed will be a random one according to a distribution of the plurality of linked documents.
12. The method of claim 1, wherein the processing includes:
displaying links to the linked documents as a directory listing.
13. The method of claim 1, wherein the processing includes:
displaying links to the linked documents, and
displaying annotations representing the score of each of the linked documents.
14. The method of claim 13, wherein the annotations are bars, icons, or text.
15. The method of claim 1, further comprising:
processing the linked documents based on textual matching.
16. The method of claim 15, wherein the textual matching includes matching anchor text associated with the links.
17. The method of claim 1, further comprising:
processing the linked documents based on groupings of the linked documents.
18. A computer-readable medium that stores instructions executable by one or more processing devices to perform a method for determining scores for a plurality of linked documents, comprising:
instructions for obtaining a plurality of documents, at least some of the documents being linked documents, at least some of the documents being linking documents, and at least some of the documents being both linked documents and linking documents, each of the linked documents being pointed to by a link in one or more of the linking documents;
instructions for determining a score for each of the linked documents based on scores for the one or more linking documents; and
instructions for processing the linked documents according to their scores.

The Rest of the Patent

It's easy to forget that the claims, and only the claims, define patent rights. The rest of the patent document is auxiliary to the claims. So remember:

- The **title** does not define the patent rights. If the title is "toaster", the patent probably covers some minor aspect of a toaster. It does not cover every aspect of every toaster, and certainly not the very concept of a toaster. It's usually best to ignore the title.
- The **drawings** do not define the patent rights. The claims probably highlight some small aspect of the drawings, and this small aspect is part of the patent rights. The rest of the drawings may help explain the claims, but that is all they do.
- The **summary** is just a summary of some technology. The claims probably highlight some small part of the summary, and this small part is part of the patent rights. The rest is fluff.

Prior Art

A prior art analysis is similar to an infringement analysis. I'll explain a quick taste of the prior art analysis because it's useful to compare it to the infringement analysis. First, check the prior art reference's date. Is it older than the patent? If so, it might be prior art. If the reference is newer than the patent, it's not prior art.

Next, skip down to the claims. As with the infringement analysis, the claims are the most important part of the patent for a prior art analysis. But unlike infringement, we will need to review all the claims, not just the independent claims.

Check to see whether the prior art reference explains *every element* of a claim in the patent. If so, the reference "anticipates" the claim, and the claim is invalid. Repeat this process for each claim. If an independent claim is anticipated and invalid, the dependent claims that branch off of it might still be valid.

The prior art reference might be another patent, or it might be a journal article, white paper, or most any other published document. If the prior art reference happens to be another patent, the claims in this prior art patent are not particularly important. Any part of a prior art patent (let's call it "old-patent") can help invalidate a later patent ("new-patent"). The claims are the critical part of new-patent, but not of old-patent.

Even if a patent claim is not *anticipated*, it might still be invalid as *obvious*. The obviousness analysis basically asks whether someone might reasonably combine two or more references to teach every element of a patent claim. It's complicated, so we will save the details for another post.

Conclusion

I hope this helps provide a general overview of how to read a patent. Patent law is complicated and full of traps for the unwary. This quick guide provides you with just enough information to be dangerous. **If you have an important patent question, hire a patent lawyer.** ■

Eric is a NYC startup and technology lawyer at Adler Vermillion LLP. He works with the Brooklyn Law school technology clinic to provide free defense against patent trolls, and sits on the board of the NYC Legal Hackers. Follow him on *@teachingaway*

Reprinted with permission of the original author.
First appeared in *hn.my/patent* (*adlervermillion*.com)

The Introduction to Reactive Programming You've Been Missing

By ANDRÉ STALTZ

So you're curious in learning this new thing called (Functional) Reactive Programming (FRP).

Learning it is hard, even harder by the lack of good material. When I started, I tried looking for tutorials. I found only a handful of practical guides, but they just scratched the surface and never tackled the challenge of building the whole architecture around it. Library documentation often doesn't help when you're trying to understand some function. I mean, honestly, look at this:

Rx.Observable.prototype.flatMapLatest(selector, [thisArg])

Projects each element of an observable sequence into a new sequence of observable sequences by incorporating the element's index and then transforms an observable sequence of observable sequences into an observable sequence producing values only from the most recent observable sequence.

Holy cow.

I've read two books, one just painted the big picture, while the other dived into how to use the FRP library. I ended up learning Reactive Programming the hard way: figuring it out

while building with it. At my work in Futurice I got to use it in a real project, and had the support of some colleagues when I ran into troubles.

The hardest part of the learning journey is **thinking in FRP**. It's a lot about letting go of old imperative and stateful habits of typical programming, and forcing your brain to work in a different paradigm. I haven't found any guide on the internet in this aspect, and I think the world deserves a practical tutorial on how to think in FRP, so that you can get started. Library documentation can light your way after that. I hope this helps you.

"What is Functional Reactive Programming (FRP)?"

There are plenty of bad explanations and definitions out there on the internet. Wikipedia is too generic and theoretical as usual. Stackoverflow's canonical answer is obviously not suitable for newcomers. Reactive Manifesto sounds like the kind of thing you show to your project manager or the businessmen at your company. Microsoft's Rx terminology "Rx = Observables + LINQ + Schedulers" is so heavy and Microsoftish that most of us are left confused. Terms like "reactive" and "propagation of change" don't convey anything specifically different to what your typical MV* and favorite language already does. Of course my framework views react to the models. Of course change is propagated. If it wouldn't, nothing would be rendered.

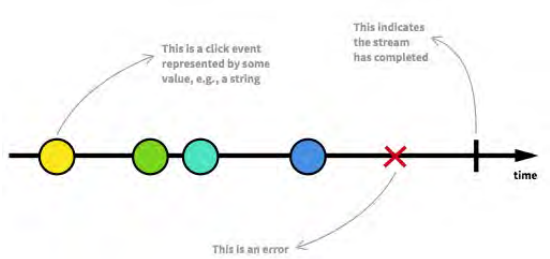
So let's cut the bullshit.

FRP is programming with asynchronous data streams.

In a way, this isn't anything new. Event buses or your typical click events are really an asynchronous event stream, on which you can observe and do some side effects. FRP is that idea on steroids. You are able to create data streams of anything, not just from click and hover events. Streams are cheap and ubiquitous, anything can be a stream: variables, user inputs, properties, caches, data structures, etc. For example, imagine your Twitter feed would be a data stream in the same fashion that click events are. You can listen to that stream and react accordingly.

On top of that, you are given an amazing toolbox of functions to combine, create and filter any of those streams. That's where the "functional" magic kicks in. A stream can be used as an input to another one. Even multiple streams can be used as inputs to another stream. You can merge two streams. You can filter a stream to get another one that has only those events you are interested in. You can map data values from one stream to another new one.

If streams are so central to FRP, let's take a careful look at them, starting with our familiar "clicks on a button" event stream.



A stream is a sequence of **ongoing events ordered in time.** It can emit three different things: a value (of some type), an error, or a "completed" signal. Consider that the "completed" takes place, for instance, when the current window or view containing that button is closed.

We capture these emitted events only **asynchronously**, by defining a function that will execute when a value is emitted, another function when an error is emitted, and another function when "completed" is emitted. Sometimes these last two can be omitted and you can just focus on defining the function for values. The "listening" to the stream is called **subscribing**. The functions we are defining are observers. The stream is the subject (or "observable") being observed. This is precisely the Observer Design Pattern.

An alternative way of drawing that diagram is with ASCII, which we will use in some parts of this tutorial:

```
--a---b-c---d---X---|->

a, b, c, d are emitted values
X is an error
| is the 'completed' signal
---> is the timeline
```

Since this feels so familiar already, and I don't want you to get bored, let's do something new: we are going to create new click event streams transformed out of the original click event stream.

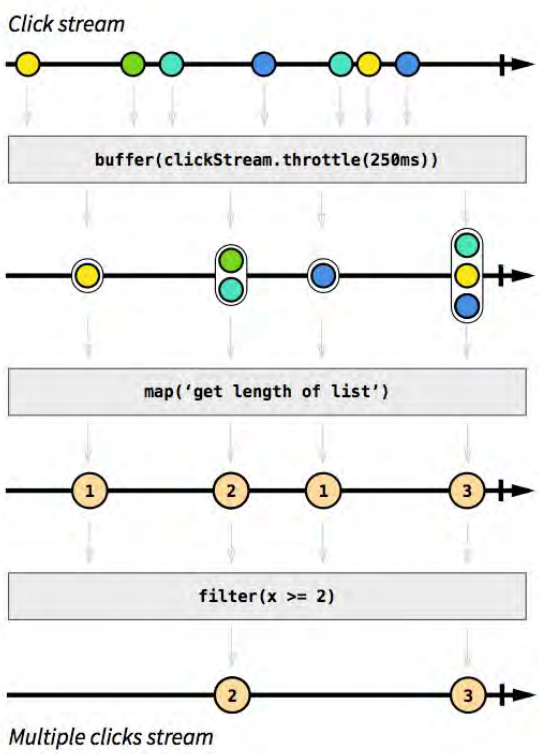
First, let's make a counter stream that indicates how many times a button was clicked. In common FRP libraries, each stream has many functions attached to it, such as `map`, `filte`, `scan`, etc. When you call one of these functions, such as `clickStream.map(f)`, it returns a new stream based on the click stream. It does not modify the original click stream in any way. This is a property called **immutability**, and it goes together with FRP streams just like pancakes are good with syrup. That allows us to chain functions like `clickStream.map(f).scan(g)`:

clickStream: ---c----c--c----c------c-->
vvvv map(c becomes 1)
vvvv
---1----1--1----1------1-->
vvvvvvvv scan(+)
vvvvvvvv
counterStream: ---1----2--3----4------5-->

The `map(f)` function replaces (into the new stream) each emitted value according to a function f you provide. In our case, we mapped to the number 1 on each click. The `scan(g)` function aggregates all previous values on the stream, producing value `x = g(accumulated, current)`, where `g` was simply the add function in this example. Then, `counterStream` emits the total number of clicks whenever a click happens.

To show the real power of FRP, let's just say that you want to have a stream of "double click" events. To make it even more interesting, let's say we want the new stream to consider triple clicks as double clicks, or in general, multiple clicks (two or more). Take a deep breath and imagine how you would do that in a traditional imperative and stateful fashion. I bet it sounds fairly nasty and involves some variables to keep state and some fiddling with time intervals.

Well, in FRP it's pretty simple. In fact, the logic is just 4 lines of code. But let's ignore code for now. Thinking in diagrams is the best way to understand and build streams, whether you're a beginner or an expert.



Grey boxes are functions transforming one stream into another. First we accumulate clicks in lists, whenever 250 milliseconds of "event silence" has happened (that's what `buffer(stream.throttle(250ms))` does, in a nutshell. Don't worry about understanding the details at this point, we are just demoing FRP for now). The result is a stream of lists, from which we apply `map()` to map each list to an integer matching the length of that list. Finally, we ignore `1` integers using the `filter(x >= 2)` function. That's it: 3 operations to produce our intended stream. We can then subscribe ("listen") to it to react accordingly how we wish.

I hope you enjoy the beauty of this approach. This example is just the tip of the iceberg: you can apply the same operations on different kinds of streams, for instance, on a stream of API responses; on the other hand, there are many other functions available.

"Why should I consider adopting FRP?"

FRP raises the level of abstraction of your code so you can focus on the interdependence of events that define the business logic, rather than having to constantly fiddle with a large amount of implementation details. Code with FRP will likely be more concise.

The benefit is more evident in modern web apps and mobile apps that are highly interactive with a multitude of UI events related to data events. Ten years ago, interaction with web pages was basically about submitting a long form to the backend and performing simple rendering to the frontend. Apps have evolved to be more real-time: modifying a single form field can automatically trigger a save to the backend, "likes" to some content can be reflected in real-time to other connected users, and so forth.

Apps nowadays have an abundance of real-time events of every kind that enable a highly interactive user experience. We need tools for properly dealing with that, and Reactive Programming is an answer.

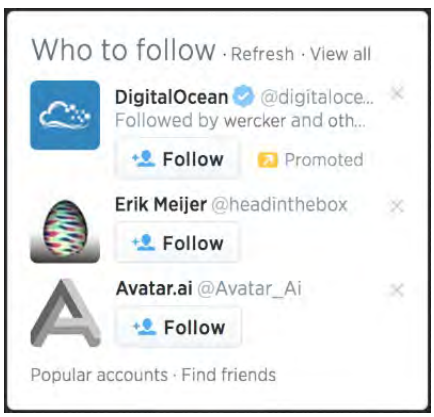
Thinking in FRP, with examples

Let's dive into the real stuff. A real-world example with a step-by-step guide on how to think in FRP. No synthetic examples, no half-explained concepts. By the end of this tutorial we will have produced real functioning code, while knowing why we did each thing.

I picked JavaScript and RxJS [hn.my/rxjs] as the tools for this for a reason: JavaScript is the most familiar language out there at the moment, and the Rx* library family [rx.codeplex.com] is widely available for many languages and platforms (.NET, Java, Scala, Clojure, JavaScript, Ruby, Python, C++, Objective-C/Cocoa, Groovy, etc.). So whatever your tools are, you can concretely benefit by following this tutorial.

Implementing a "Who to follow" suggestions box

In Twitter there is this UI element that suggests other accounts you could follow:



We are going to focus on imitating its core features, which are:

- On startup, load accounts data from the API and display 3 suggestions
- On clicking "Refresh," load 3 other account suggestions into the 3 rows
- On click "x" button on an account row, clear only that current account and display another
- Each row displays the account's avatar and links to their page

We can leave out the other features and buttons because they are minor. And, instead of Twitter, which recently closed its API to the unauthorized public, let's build that UI for following people on Github. There's a Github API for getting users.

Request and response

How do you approach this problem with FRP? Well, to start with, (almost) *everything can be a stream*. That's the FRP mantra. Let's start with the easiest feature: "on startup, load 3 accounts data from the API." There is nothing special here, this is simply about (1) doing a request, (2) getting a response, and (3) rendering the response. So let's go ahead and represent our requests as a stream. At first this will feel like overkill, but we need to start from the basics, right?

On startup we need to do only one request, so if we model it as a data stream, it will be a stream with only one emitted value. Later, we know we will have many requests happening, but for now, it is just one.

```
--a------|->

Where a is the string 'https://
api.github.com/users'
```

This is a stream of URLs that we want to request. Whenever a request event happens, it tells us two things: when and what. "When" the request should be executed is when the event is emitted. And "what" should be requested is the value emitted: a string containing the URL.

To create such stream with a single value is very simple in Rx*. The official terminology for a stream is "Observable," for the fact that it can be observed, but I find it to be a silly name, so I call it stream.

```
var requestStream = Rx.Observable.
returnValue('https://api.github.
com/users');
```

But now, that is just a stream of strings, doing no other operation, so we need to somehow make something happen when that value is emitted. That's done by subscribing to the stream.

```
requestStream.
subscribe(function(requestUrl) {
  // execute the request
  jQuery.getJSON(requestUrl,
function(responseData) {
    // ...
  });
}
```

Notice we are using a jQuery Ajax callback (which we assume you should know already) to handle the asynchronicity of the request operation. But wait a moment, FRP is for dealing with **asynchronous** data streams. Couldn't the response for that request be a stream containing the data arriving at some time in the future? Well, at a conceptual level, it sure looks like it, so let's try that.

```
requestStream.
subscribe(function(requestUrl) {
  // execute the request
  var responseStream =
Rx.Observable.create(function
(observer) {
    jQuery.getJSON(requestUrl)
    .done(function(response) {
observer.onNext(response); })
    .fail(function(jqXHR, status,
error) { observer.onError(error);
})
    .always(function() { observer.
onCompleted(); });
  });
  responseStream.
subscribe(function(response) {
    // do something with the
    // response
  });
}
```

What `Rx.Observable.create()` does is create your own custom stream by explicitly informing each observer (or in other words, a "subscriber") about data events (`onNext()`) or errors (`onError()`). What we did was just wrap that jQuery Ajax Promise. **Excuse me, does this mean that a Promise is an Observable?**

Yes.

Observable is Promise++. In Rx you can easily convert a Promise to an Observable by doing `var stream = Rx.Observable.fromPromise(promise)`, so let's use that. The only difference is that Observables are not Promises/A+ compliant, but conceptually there is no clash. A Promise is simply an Observable with one single emitted value. FRP streams go beyond promises by allowing many returned values.

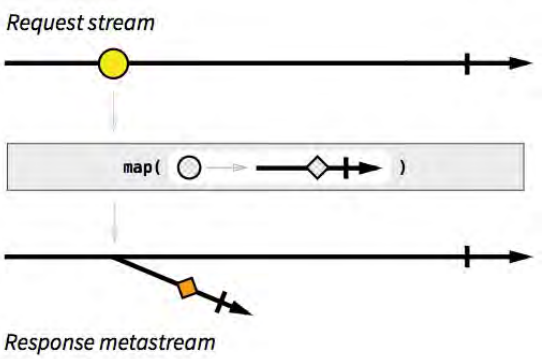
This is pretty nice, and shows how FRP is at least as powerful as Promises. So if you believe the Promises hype, keep an eye on what FRP is capable of.

Now back to our example, if you were quick to notice, we have one `subscribe()` call inside another, which is somewhat akin to callback hell. Also, the creation of `responseStream` is dependent on `requestStream`. As you heard before, in FRP there are simple mechanisms for transforming and creating new streams out of others, so we should be doing that.

The one basic function that you should know by now is `map(f)`, which takes each value of stream A, applies `f()` on it, and produces a value on stream B. If we do that to our request and response streams, we can map request URLs to response Promises (disguised as streams).

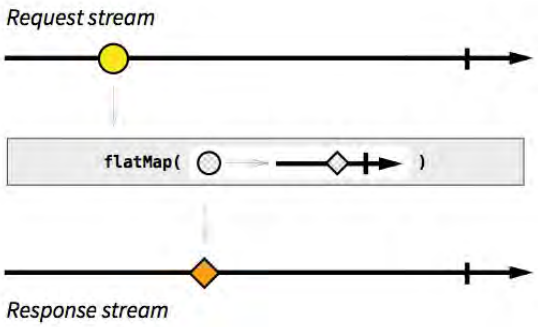
```
var responseMetastream = requestStream
  .map(function(requestUrl) {
    return Rx.Observable.fromPromise(jQuery.getJSON(requestUrl));
  });
```

Then we will have created a beast called *metastream*, a stream of streams. Don't panic yet. A metastream is a stream where each emitted value is yet another stream. You can think of it as pointers: each emitted value is a pointer to another stream. In our example, each request URL is mapped to a pointer to the promise stream containing the corresponding response.



A metastream for responses looks confusing, and doesn't seem to help us at all. We just want a simple stream of responses, where each emitted value is a JSON object, not a "Promise" of a JSON object. Say hi to Mr. Flatmap: a version of `map()` than "flattens" a metastream by emitting on the "trunk" stream everything that will be emitted on "branch" streams. Flatmap is not a "fix" and metastreams are not a bug; these are really the tools for dealing with asynchronous responses in FRP.

```
var responseStream = requestStream
  .flatMap(function(requestUrl) {
    return Rx.Observable.fromPromise(jQuery.getJSON(requestUrl));
  });
```



Nice. And because the response stream is defined according to request stream, **if** we have more events happening on request stream later on, we will have the corresponding response events happening on response stream, as expected:

```
requestStream:
--a-----b--c------------|->
responseStream:
-----A--------B-----C---|->

(lowercase is a request, uppercase
is its response)
```

Now that we finally have a response stream, we can render the data we receive:

```
responseStream.
subscribe(function(response) {
  // render `response` to the DOM
  // however you wish
});
```

Joining all the code until now, we have:

```
var requestStream = Rx.Observable.
returnValue('https://api.github.
com/users');

var responseStream = requestStream
  .flatMap(function(requestUrl) {
    return Rx.Observable.
fromPromise(jQuery.
getJSON(requestUrl));
  });

responseStream.
subscribe(function(response) {
  // render `response` to the DOM
  // however you wish
});
```

The refresh button

I did not yet mention that the JSON in the response is a list with 100 users. The API only allows us to specify the page offset, and not the page size, so we're using just 3 data objects and wasting 97 others. We can ignore that problem for now, since later on we will see how to cache the responses.

Every time the refresh button is clicked, the request stream should emit a new URL, so that we can get a new response. We need 2 things: a stream of click events on the refresh button (mantra: anything can be a stream), and we need to change the request stream to depend on the refresh click

stream. Gladly, RxJS comes with tools to make Observables from event listeners.

```
var refreshButton = document.que-
rySelector('.refresh');
var refreshClick-
Stream = Rx.Observable.
fromEvent(refreshButton, 'click');
```

Since the refresh click event doesn't itself carry any API URL, we need to map each click to an actual URL. Now we change the request stream to be the refresh click stream mapped to the API endpoint with a random offset parameter each time.

```
var requestStream = refreshClick-
Stream
  .map(function() {
    var randomOffset = Math.
floor(Math.random()*500)
    return 'https://api.github.
com/users?since=' + randomOffset;
  });
```

Because I'm dumb and I don't have automated tests, I just broke one of our previously built features. A request doesn't happen anymore on startup; it happens only when the refresh is clicked. Urgh. I need both behaviors: a request when either a refresh is clicked or the webpage was just opened.

We know how to make a separate stream for each one of those cases:

```
var requestOnRefreshStream =
refreshClickStream
  .map(function() {
    var randomOffset = Math.
floor(Math.random()*500)
    return 'https://api.github.
com/users?since=' + randomOffset;
  });

var startupRequest-
Stream = Rx.Observable.
returnValue('https://api.github.
com/users');
```

But how can we "merge" these two into one? Well, there's `merge()`. Explained in the diagram dialect, this is what it does:

```
stream A:
---a--------e-----o----->
stream B:
-----B---C-----D-------->

vvvvvvvvv merge vvvvvvvvv

---a-B---C--e--D--o----->
```

It should be easy now:

```
var requestOnRefreshStream =
refreshClickStream
  .map(function() {
    var randomOffset = Math.
floor(Math.random()*500)
    return 'https://api.github.
com/users?since=' + randomOffset;
  });

var startupRequest-
Stream = Rx.Observable.
returnValue('https://api.github.
com/users');

var requestStream = Rx.Observable.
merge(
  requestOnRefreshStream, startu-
pRequestStream
);
```

There is an alternative and cleaner way of writing that, without the intermediate streams.

```
var requestStream = refreshClick-
Stream
  .map(function() {
    var randomOffset = Math.
floor(Math.random()*500)
    return 'https://api.github.
com/users?since=' + randomOffset;
  })
  .merge(Rx.Observable.
returnValue('https://api.github.
com/users'));
```

Even shorter, even more readable:

```
var requestStream = refreshClick-
Stream
  .map(function() {
    var randomOffset = Math.
floor(Math.random()*500)
    return 'https://api.github.
com/users?since=' + randomOffset;
  })
  .startWith('https://api.github.
com/users');
```

The `startWith()` function does exactly what you think it does. No matter how your input stream looks like, the output stream resulting of `startWith(x)` will have x at the beginning. But I'm not DRY enough, I'm repeating the API endpoint string. One way to fix this is by moving the `startWith()` close to the `refreshClickStream`, to essentially "emulate" a refresh click on startup.

```
var requestStream = refreshClick-
Stream.startWith('startup click')
  .map(function() {
    var randomOffset = Math.
floor(Math.random()*500)
    return 'https://api.github.
com/users?since=' + randomOffset;
  });
```

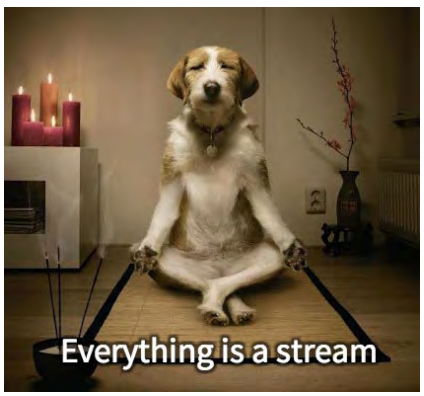
Nice. If you go back to the point where I "broke the automated tests," you should see that the only difference with this last approach is that I added the `startWith()`.

Modeling the 3 suggestions with streams

Until now, we have only touched a suggestion UI element on the rendering step that happens in the `responseStream`'s `subscribe()`. Now with the refresh button, we have a problem: as soon as you click "refresh," the current 3 suggestions are not cleared. New suggestions come in only after a response has arrived, but to make the UI look nice, we need to clean out the current suggestions when clicks happen on the refresh.

```
refreshClickStream.
subscribe(function() {
  // clear the 3 suggestion DOM
  // elements
});
```

No, not so fast, pal. This is bad, because we now have 2 subscribers that affect the suggestion DOM elements (the other one being `responseStream.subscribe()`), and that doesn't really sound like Separation of concerns. Remember the FRP mantra?



So let's model a suggestion as a stream, where each emitted value is the JSON object containing the suggestion data. We will do this separately for each of the 3 suggestions. This is how the stream for suggestion #1 could look like:

```
var suggestion1Stream = respons-
eStream
  .map(function(listUsers) {
    // get one random user from
    // the list
    return listUsers[Math.
floor(Math.random()*listUsers
length)];
  });
```

The others, `suggestion2Stream` and `suggestion3Stream` can be simply copied and pasted from `suggestion-1Stream`. This is not DRY, but it will keep our example simple for this tutorial. Plus, I think it's a good exercise on how to avoid repetition in this case.

Instead of having the rendering happen in `responseStream`'s `subscribe()`, we do that here:

```
suggestion1Stream.
subscribe(function(suggestion) {
  // render the 1st suggestion to
  // the DOM
});
```

Back to the "on refresh, clear the suggestions," we can simply map refresh clicks to `null` suggestion data, and include that in the `suggestion1Stream`, as such:

```
var suggestion1Stream = respons-
eStream
  .map(function(listUsers) {
    // get one random user from
    // the list
    return listUsers[Math.
floor(Math.random()*listUsers
length)];
  })
  .merge(
    refreshClickStream.
map(function(){ return null; })
  );
```

And when rendering, we interpret `null` as "no data," hence hiding its UI element.

```
suggestion1Stream.
subscribe(function(suggestion) {
  if (suggestion === null) {
    // hide the first suggestion
   // DOM element
  }
  else {
    // show the first suggestion
    // DOM element
    // and render the data
  }
});
```

The big picture is now:

```
refreshClickStream:
----------o--------o---->
     requestStream:
-r--------r--------r---->
    responseStream:
----R---------R------R-->
 suggestion1Stream:
----s-----N---s----N-s-->
 suggestion2Stream:
----q-----N---q----N-q-->
 suggestion3Stream:
----t-----N---t----N-t-->
```

Where `N` stands for `null`.

As a bonus, we can also render "empty" suggestions on startup. That is done by adding `startWith(null)` to the suggestion streams:

```
var suggestion1Stream = respons-
eStream
  .map(function(listUsers) {
    // get one random user from
    // the list
    return listUsers[Math.
floor(Math.random()*listUsers
length)];
  })
  .merge(
    refreshClickStream.
map(function(){ return null; })
  )
  .startWith(null);
```

Which results in:

```
refreshClickStream:
----------o---------o---->
     requestStream:
-r--------r---------r---->
    responseStream:
----R----------R------R-->
 suggestion1Stream:
-N--s-----N----s----N-s-->
 suggestion2Stream:
-N--q-----N----q----N-q-->
 suggestion3Stream:
-N--t-----N----t----N-t-->
```

Closing a suggestion and using cached responses

There is one feature remaining to implement. Each suggestion should have its own "x" button for closing it, and loading another in its place. At first thought, you could say it's enough to make a new request when any close button is clicked:

```
var close1Button = document.que-
rySelector('.close1');
var close1Click-
Stream = Rx.Observable.
fromEvent(close1Button, 'click');
// and the same for close2Button
// and close3Button

var requestStream = refreshClick-
Stream.startWith('startup click')
  .merge(close1ClickStream) // we
added this
  .map(function() {
    var randomOffset = Math.
floor(Math.random()*500)
    return 'https://api.github.
com/users?since=' + randomOffset;
  });
```

That does not work. It will close and reload *all* suggestions, rather than just only the one we clicked on. There are a couple of different ways of solving this, and to keep it interesting, we will solve it by reusing previous responses. The API's response page size is 100 users while we were using just 3 of those, so there is plenty of fresh data available. No need to request more.

Again, let's think in streams. When a "close1" click event happens, we want to use the *most recently emitted* response on `responseStream` to get one random user from the list in the response. As such:

```
    requestStream:
--r--------------->
   responseStream:
------R----------->
close1ClickStream:
------------c----->
suggestion1Stream:
------s-----s----->
```

In Rx* there is a combinator function called `combineLatest` that seems to do what we need. It takes two streams A and B as inputs, and whenever either stream emits a value, `combineLatest` joins the 2 most recently emitted

values a and b from both streams and outputs a value c = f(x,y), where f is a function you define. It is better explained with a diagram:

```
stream A:
--a-----------e--------i-------->
stream B:
-----b----c--------d-------q---->

vvvvvvvv combineLatest(f) vvvvvvv

----AB---AC--EC---ED--ID--IQ---->

where f is the uppercase function
```

We can apply combineLatest() on close1ClickStream and responseStream, so that whenever the close 1 button is clicked, we get the latest response emitted and produce a new value on suggestion1Stream. On the other hand, combineLatest() is symmetric: whenever a new response is emitted on responseStream, it will combine with the latest "close 1" click to produce a new suggestion. That is interesting, because it allows us to simplify our previous code for suggestion1Stream, like this:

```
var suggestion1Stream = close-
1ClickStream
  .combineLatest(responseStream,
    function(click, listUsers) {
      return listUsers[Math.
floor(Math.random()*listUsers
length)];
    })
  .merge(
    refreshClickStream.
map(function(){ return null; }))
  .startWith(null);
```

One piece is still missing in the puzzle. The combineLatest() uses the most recent of the two sources, but if one of those sources hasn't emitted anything yet, combineLatest() cannot produce a data event on the output stream. If you look at the ASCII diagram above, you will see that the output has nothing when the first stream emitted value a. Only when the second stream emitted value b could it produce an output value.

There are different ways of solving this, and we will stay with the simplest one, which is simulating a click to the "close 1" button on startup:

```
var suggestion1Stream = close-
1ClickStream.startWith('startup
click') // we added this
  .combineLatest(responseStream,
    function(click, listUsers) {l
      return listUsers[Math.
floor(Math.random()*listUsers
length)];
    }
  )
  .merge(
    refreshClickStream.
map(function(){ return null; })
  )
  .startWith(null);
```

Wrapping up

And we're done. The complete code for all this was:

```
var refreshButton = document.que-
rySelector('.refresh');
var refreshClick-
Stream = Rx.Observable.
fromEvent(refreshButton, 'click');

var closeButton1 = document.que-
rySelector('.close1');
var close1Click-
Stream = Rx.Observable.
fromEvent(closeButton1, 'click');
// and the same logic for close2
// and close3

var requestStream = refreshClick-
Stream.startWith('startup click')
  .map(function() {
    var randomOffset = Math.
floor(Math.random()*500)
    return 'https://api.github.
com/users?since=' + randomOffset;
  });

var responseStream = requestStream
  .flatMap(function (requestUrl) {
    return Rx.Observable.fromPro-
mise($.ajax({url: requestUrl}));
  });

var suggestion1Stream = close-
1ClickStream.startWith('startup
click')
  .combineLatest(responseStream,
    function(click, listUsers) {
      return listUsers[Math.
floor(Math.random()*listUsers
length)];
    })
  .merge(
    refreshClickStream.
map(function(){ return null; })
  )
  .startWith(null);
// and the same logic for
// suggestion2Stream and
// suggestion3Stream

suggestion1Stream.
subscribe(function(suggestion) {
  if (suggestion === null) {
    // hide the first suggestion
    // DOM element
  }
  else {
    // show the first suggestion
    // DOM element
    // and render the data
  }
});
```

You can see this working example at *http://jsfiddle.net/staltz/8jFJH/48/*

That piece of code is small but dense: it features management of multiple events with proper separation of concerns, and even caching of responses. The functional style made the code look more declarative than imperative: we are not giving a sequence of instructions to execute, we are just

telling what something is by defining relationships between streams. For instance, with FRP we told the computer that *`suggestion1Stream` **is** the "close 1" stream combined with one user from the latest response, besides being `null` when a refresh happens or program startup happened*.

Notice also the impressive absence of control flow elements such as `if`, `for`, `while`, and the typical callback-based control flow that you expect from a JavaScript application. You can even get rid of the `if` and `else` in the `subscribe()` above by using `filter()`if you want (I'll leave the implementation details to you as an exercise). In FRP, we have stream functions such as `map`, `filte`, `scan`, `merge`, `combineLatest`, `startWith`, and many more to control the flow of an event-driven program. This toolset of functions gives you more power in less code.

What comes next

If you think Rx* will be your preferred library for Reactive Programming, take a while to get acquainted with the big list of functions for transforming, combining, and creating Observables. If you want to understand those functions in diagrams of streams, take a look at RxJava's very useful documentation with marble diagrams [hn.my/observables]. Whenever you get stuck trying to do something, draw those diagrams, think on them, look at the long list of functions, and think more. This workflow has been effective in my experience.

Once you start getting the hang of programming with Rx*, it is absolutely required to understand the concept of Cold vs Hot Observables [hn.my/coldhot]. If you ignore this, it will come back and bite you brutally. You have been warned. Sharpen your skills further by learning real functional programming, and getting acquainted with issues such as side effects that affect Rx*.

FRP works great for event-heavy frontends and apps. But it is not just a client-side thing, it works great also in the backend and close to databases. In fact, RxJava is a key component for enabling server-side concurrency in Netflix's API. FRP is not a framework restricted to one specific type of application or language. It really is a paradigm that you can use when programming any event-driven software. ■

André is a frontend developer and designer, founder of Iroquote.com and consultant at *Futurice.com*. He is MSc in Cloud Computing and BSc in Computational Mathematics. André discovered Rx as a sweet spot between his event-driven development skills and theoretical knowledge.

Reprinted with permission of the original author. First appeared in *hn.my/rx* (github.com)

A Scientist Stole my Root Beer

By JUSTIN BROWER

I'm not ashamed to say that I like root beer. There's something about the herbal and woodsy flavors that I enjoy. It's refreshing, but also a time machine. A sip of root beer brings me back to my childhood and makes me feel like a little kid again. More so than any other drink, it is classic Americana. And if I ever have the option of drinking root beer, I'll choose it over any another soda. But as much as I love root beer, there's one thing that really grinds my gears, gets my goat, and burns my bacon: A scientist stole my root beer.

Root beer has, in some way, shape, or form, been around for centuries. People undoubtedly mixed roots, berries, and herbs together in water to create teas and elixirs, either to make polluted water more palatable or as some sort of remedy. Throw in some sugar to make it go down easier and a little local yeast from, well, everywhere, and voila, you've got fermentation and beer. These types of drinks were popular in colonial America, and called "small beers" because of their low alcohol content, around 2%.

Sassafras albidum leaves

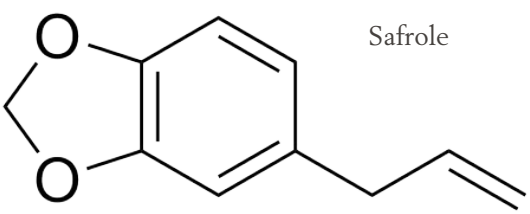
But the man that gets the glory for "inventing" root beer is Charles Hires, a pharmacist with an entrepreneurial spirit. Legend has it he was on his honeymoon and came across a tea that he particularly liked. Upon his return home he replicated the recipe and sold it as a "cure-all" elixir, which were all the rage at the time. His concoction was originally called root tea, but he renamed it root beer shortly before he displayed it at the 1876 Centennial Exposition*, supposedly to make it more appealing to the working class. Hey, it works for me. His genius came in not just selling root beer, but marketing it, and selling kits so that people could brew their own at home. Then in 1884 he made a liquid concentrate, a.k.a. syrup, so that people could skip the brewing process and "just add water." This is still how we deliver and sell root beer and other sodas today.

So what's the "root" in root beer? That would be the roots and bark of the Sassafras tree, which was used in making America's iconic root beer all the way up to 1960. Native to the Eastern United States, Sassafras albidum is a deciduous tree of medium height (~30 feet) that is often grown for its ornamental appearance and fragrance. In the fall the leaves turn spectacular shades of red and orange. In the woodsy wild, Sassafras trees are easy to identify because the leaves are

shaped like mittens. Seriously. You've got a left and right handed mitten, and a double mitten, like for people with two thumbs on each hand. Just look at the photo (on the right), it's easier that way.

The fragrance of Sassafras comes from essential oils present in the roots and bark of the tree. Notable chemicals include aromatic compounds like α-pinene (pine scented, duh) and camphor (Vicks VapoRub) as well as possible hallucinogens thujone and ,myristicin which you've read about here, of course. But the chemical getting all the glory, or the blame, is safrole — and if you remember your myristicin, you'd see that they look a lot alike.

Safrole is the primary constituent of Sassafras oil, but when reading about safrole I see a lot of bad math, which I think propagates itself into other write-ups — without references of course. But here's what I came up with, which is the absolute best case scenario for determining safrole concentration in Sassafras root, but also not an indicator of what you'd find in a steeped tea. I'll explain.



Safrole

From 150 grams of ground Sassafras root a total of 0.68 grams (680 milligrams) of safrole was extracted using 3 liters of petrol, a low boiling mixture of hydrocarbons that dissolves non-polar (read: greasy) things, like safrole, giving a total percent yield of 0.4% safrole. And that's coming from two sets of extracts: one that yields 0.44 grams of oil that is 90% safrole and one which is 4.87 grams of oil but only 6% safrole. What I often read is along the lines of: Sassafras contains ~3% essential oil of which 90% is safrole. It's true that there is ~3% essential oil, but only one extract in this case is 90% safrole. If you do the math, you'd calculate that safrole makes up 2.8% of the total extracted oil. So right off the bat, people estimate safrole almost 10-times too high.

Enough math. The point I really want to make is that this is a best case scenario and not representative of a real world scenario. First, the above experiment is using ground Sassafras. And when I say ground I mean like coffee grounds. Second, they are extracting out safrole (and other oils) using a non-polar solvent. They have to, because safrole is insoluble in water. It is literally like oil in water. I don't know how you make your Sassafras tea, but most people don't have the ability to grind wood into a powder and they sure as hell aren't mixing it with gasoline. So the amount of safrole extracted from

small chunks of root in hot water? I don't know, but I can guarantee you it's much, much less than 0.4%, and likely more along the lines of 0.04%.

I'm actually getting a bit worked up. Can you tell? So why my fixation on safrole? Because some jackass thought it would be a good idea to feed it to rats and see what happens. And if your agenda is to show that safrole is toxic, feed them huge amounts. Like 0.5 grams per kilogram of body weight every day for 2 months. To obtain this much safrole naturally, the rat would have to eat its body weight in sassafras root every day. What happened? They established an LD50 (lethal dose for 50% of the population) in rats of 1950 mg/kg. They also saw liver damage and tumor formation, at high doses, due to safrole-DNA adducts attributed to a metabolite found in rats, 1'-hydroxysafrole.

Liver damage and cancer is bad. Period. But obviously I've got some issues, at least that's what everyone tells me. I hate the "scale-up game" between rats and humans. It just doesn't work. Rats and humans aren't the same, and we don't metabolize things the same. With that said, to just give you an idea of the magnitude we're talking about, if I wanted to consume 0.5 grams/kg of safrole, and assuming I can extract out 0.04% safrole from Sassafras root chunks in water, I'd need to boil up 170 pounds of sassafras root…in about 400 gallons of water. If I had to dig up 170 pounds of Sassafras root every day for 2 months I'd die from exhaustion long before the cancer got me. Now obviously you don't want liver damage. Or cancer. I don't even want an LD1, let alone an LD50. But hopefully you can see how ridiculous this is. But the best part is that hepatocarcinogenic metabolite, 1′ -hydroxysafrole. Remember that one, the one that messes with the DNA? Well, it's not even found in humans. Really? Seriously.

So we're left with a chemical that's insoluble in water, in already low concentrations, that causes damage in rats at obscenely high amounts via a metabolite not even found in man. What's the U.S. government to do? Ban it of course. In 1960 the U.S. Food and Drug Administration banned the addition of safrole in foods. Never mind that safrole is present in everyday foods like cinnamon and basil. This bone-headed decision forced root beer brewers to abandon Sassafras roots and extracted oils, and instead turn toward other additives to make up for the flavor loss.

I know this is getting long, but there are two conspiracy theories out there regarding why the FDA banned safrole, and you know I love me some conspiracy theories:

1. Cola companies, particularly Coca-Cola, was concerned that root beer was cutting into their sales and profit margins, and coerced the FDA into banning safrole to put the hurt on the brewers. This I could buy into, because Coca-Cola has butted heads with the FDA before. Around 1910 the FDA wanted Coca-Cola to stop adding caffeine to their products, and even sued them. Coca-Cola said no, flipped them the bird, and went about their merry way. So there is some history of big business having power over government.
2. Safrole is a building block in the synthesis of MDMA, also known worldwide as Ecstasy. In two easy steps (or less if you're clever), you can synthesize a whole range of MDMA and related designer stimulant drugs. This has the negative effect of massive deforestation in Asian countries of safrole containing trees, with a large portion of it being funneled towards illegal MDMA manufacturing in China and the U.S. The problem with this theory though is that although MDMA has been known since the early 1900's, and tested in humans in the '50s, it wasn't used as a recreational drug until the late Alexander Shulgin's lab synthesized and tried it out in the early '80s. Then in 1985 the DEA scheduled MDMA as a schedule-1 drug. So the timing is a bit off for the FDA to become involved in the MDMA scene.

Throw in the fact that I can't find any cases of people becoming ill, let alone developing cancer from drinking root beer or tea made from Sassafras root, despite being used for centuries, makes me think there were either some shenanigans going on at the FDA or some really bad science. I vote for bad science . . . with a dash of conspiracy.

So what does "real" root beer taste like? I have no idea…some scientist stole it from me. ■

Justin Brower, Ph.D. is a forensic toxicologist and chemist living and working in North Carolina. He combined his love of chemistry with things that kill and started *NaturesPoisons.com*, where he writes about the history and science of poisons, venoms and toxins.

Reprinted with permission of the original author.
First appeared in *hn.my/rootbeer* (naturespoisons.com)