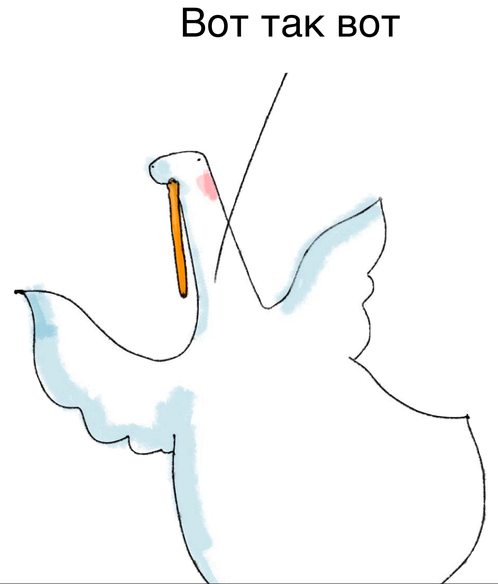
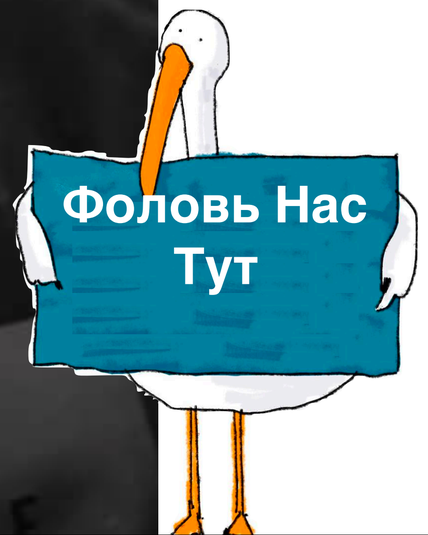
The State of Reactive Streams





netifi

Socket

JEE Conf

DEVOXX UKRAINE

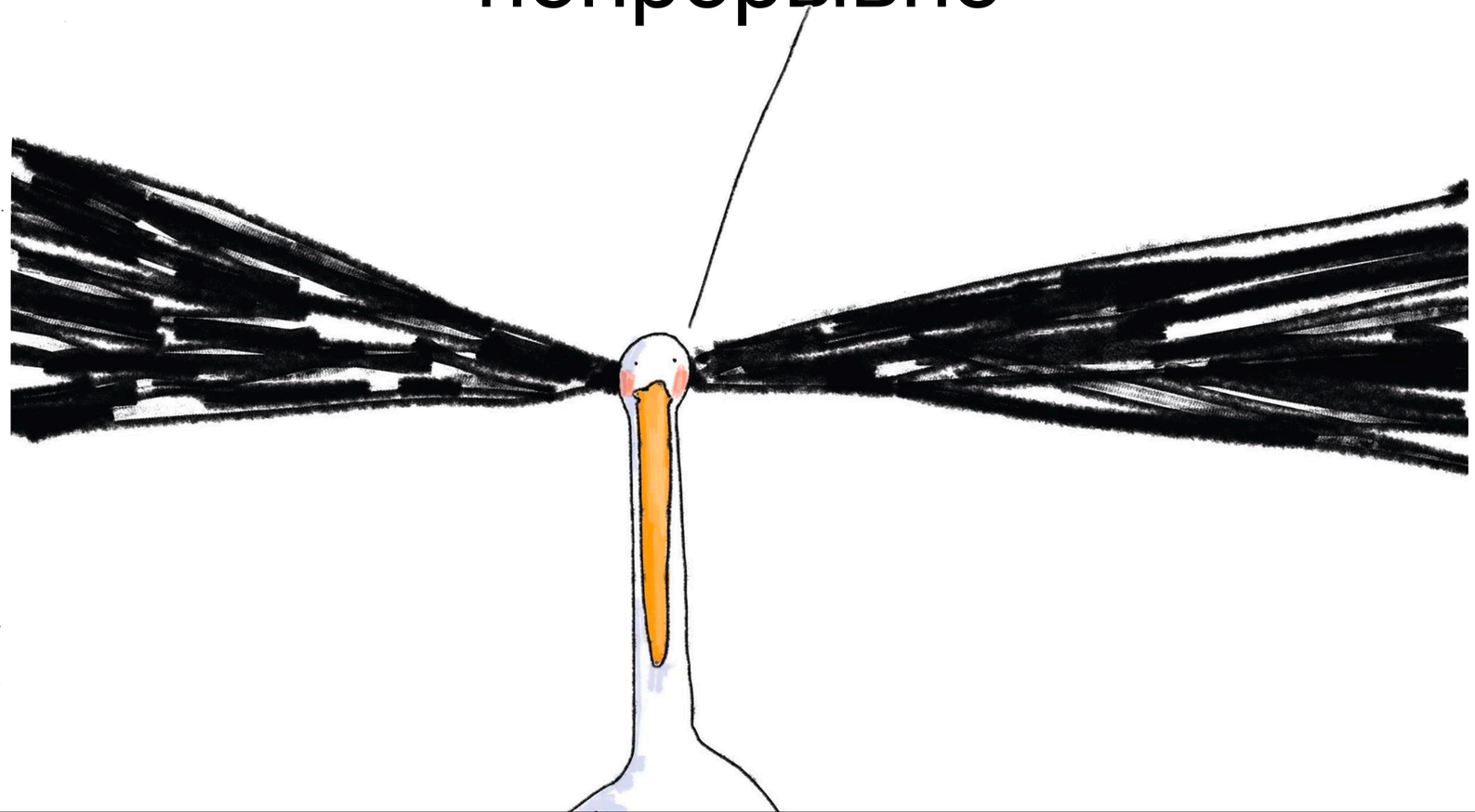
JUG.UA

Что будет

- Зачем нам нужна реактивщина и как она возникла
- Что у нас есть сегодня?
- Что ждет в будущем?

Это все о взаимодействии

Реактивная Система - тип систем, реагирующий на все события непрерывно



On the Development of Reactive Systems*

D. Harel and A. Pnueli

Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot 76100, Israel

January, 1985

Abstract

Some observations are made concerning the process of developing complex systems. A broad class of systems, termed *reactive*, is singled out as being particularly problematic when it comes to finding satisfactory methods for behavioral description. In this paper we recommend the recently proposed statechart method for this purpose. Moreover, it is observed that most reactive systems cannot be developed in a linear stepwise fashion, but, rather, give rise to a two-dimensional development process, featuring behavioral aspects in the one dimension and implementational ones in the other. Concurrency may occur in both dimensions, as *orthogonality* of states in the one and as *parallelism* of subsystems in the other. A preliminary approach to working one's way through this "magic square" of system development is then presented. The ideas described herein seem to be relevant to a wide variety of application areas.

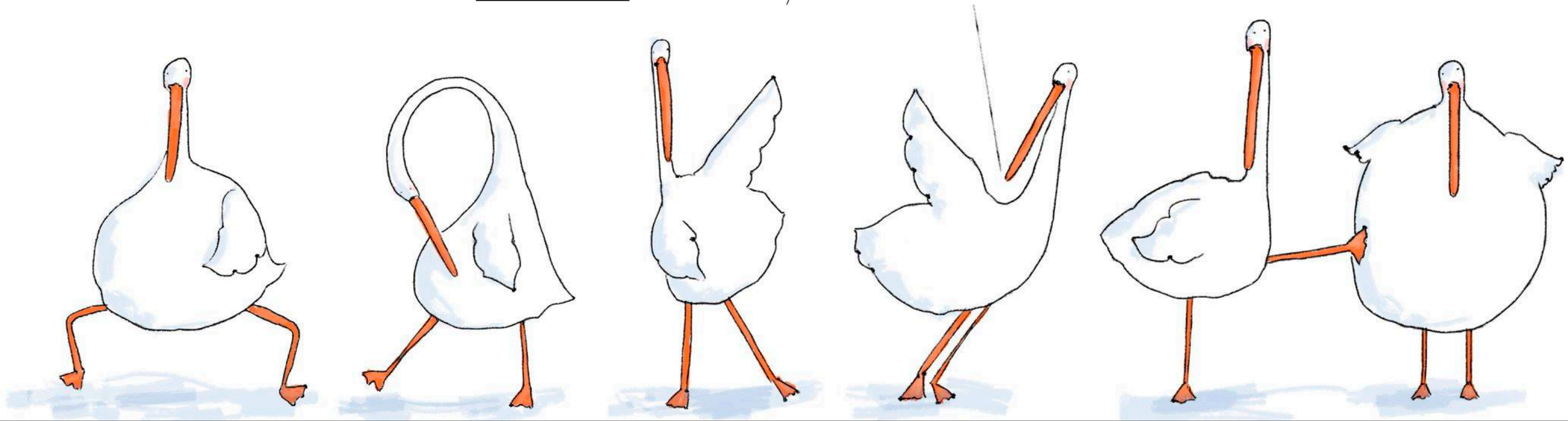
Why Another Paper on System Development?

The literature on software engineering, programming languages, and system and hardware design, is brimming with papers describing methods for specifying and designing large and complex systems. Why then are we

Отличия от не реактивных систем

- **НЕ** реактивные системы - обрабатывают события **выборочно** и **не своевременно**

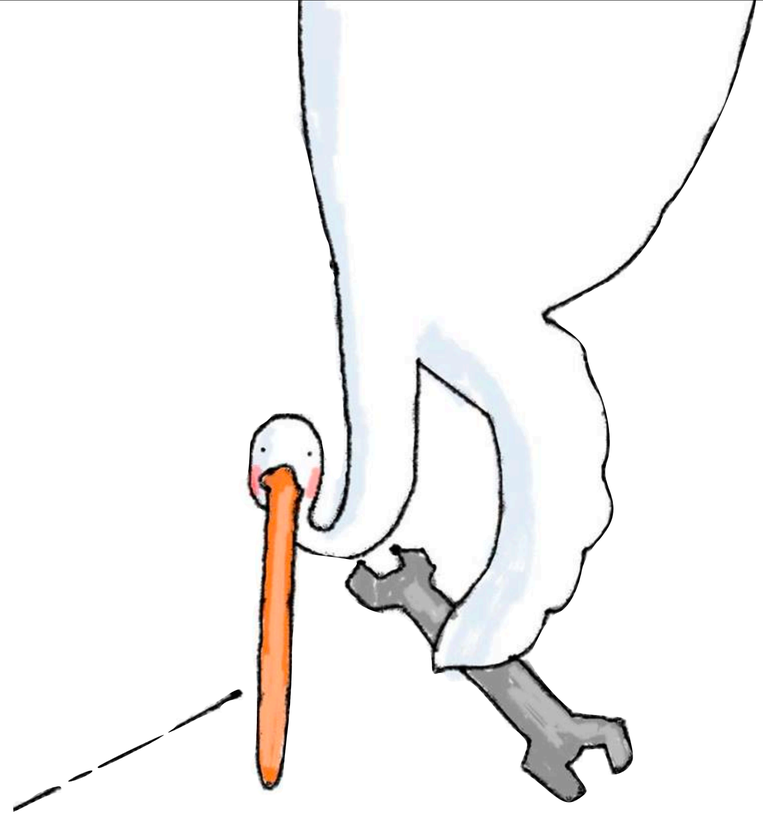
- **Реактивная** система - изменяет свое **поведение** как **результат** обработки **всех** произошедших **событий** (**реакция** на все входящие события)



Все начало с UI еще в 90х

Ну приветики

Проблематика



Для построения отзывчивого интерфейса необходимы соответсвующие подходы


Демка

The Windows Programming Experience

by Charles Petzold and Richard Hale Shaw

Some developers think of Microsoft Windows programming as an exciting adventure; others consider it a nightmare. In truth, it's a little of both.

If you'll take the time to learn its ins and outs, programming for Windows makes possible attractive, interactive programs with a consistent user interface, device-independent graphics, and rudimentary interprocess communication and data sharing.

Most professional Windows developers use an industrial-strength C compiler (such as Microsoft C/C++, Version 7.0, or Borland C++, Version 3.1), although many smaller companies also offer C compilers suitable for Windows development (see our July 1992 stories on C/C++ compilers). In addition, Microsoft and Borland have created entry-level versions of their C development environments (Microsoft Quick C and Borland's Turbo C++) for Windows. Pascal fans can turn to Borland's Turbo Pascal for Windows. All of these packages include the additional files and tools (such as header files and resource-editing utilities) necessary for Windows programming.

With the extensions to Windows for multimedia and pen computing, Windows programmers now have access to over 1,000 function calls with descriptive names such as CreateWindow and Ellipse. These function calls are located in the DLLs (dynamic link libraries) that make up Windows. DLLs are libraries of procedures that are linked with programs at runtime and can be accessed by several programs at once.

Although dealing with these thousand function calls can be daunting, that's not usually the biggest obstacle to novice Windows programmers. Windows turns traditional structured programming inside out by being based on an "event-driven" (or "message-driven") architecture. In a traditional program, the program calls the operating system when it needs a task carried out. In an event-driven program, the operating system calls the program to inform it of important events.

Many of these events originate from user input through the keyboard or mouse. But Windows is structured so that programs can handle higher-level events that result from user input, such as selecting a menu command.

Architecturally, Microsoft Windows is based on objects called *windows* — generally rectangular areas on the screen that receive user input and display graphical output. A program usually has a main window; dialog boxes are separate windows, and all the controls in dialog boxes (buttons, list boxes, scroll bars) are also windows.

Windows turns traditional structured programming inside out by being based on an "event-driven" architecture.

Each window is associated with a *window procedure.* This is a function that may be located in a Windows program or a Windows DLL. The window procedure is called from Windows to notify it of events that effect the window. These events take the form of *messages.* A window procedure can do something with a message or pass it back to Windows for default processing. For a main application window, for example, default processing within Windows handles all the logic that is associated with menu display and selection. User-input messages are translated into menu-command messages so that the window procedure will be able to handle higher-level messages instead.

This architecture allows specific functionality to be encapsulated into different types of windows. Every window is created based on a *window class* associated with a unique window procedure. Thus, a single window procedure inside Windows handles all buttons that appear in Windows programs.

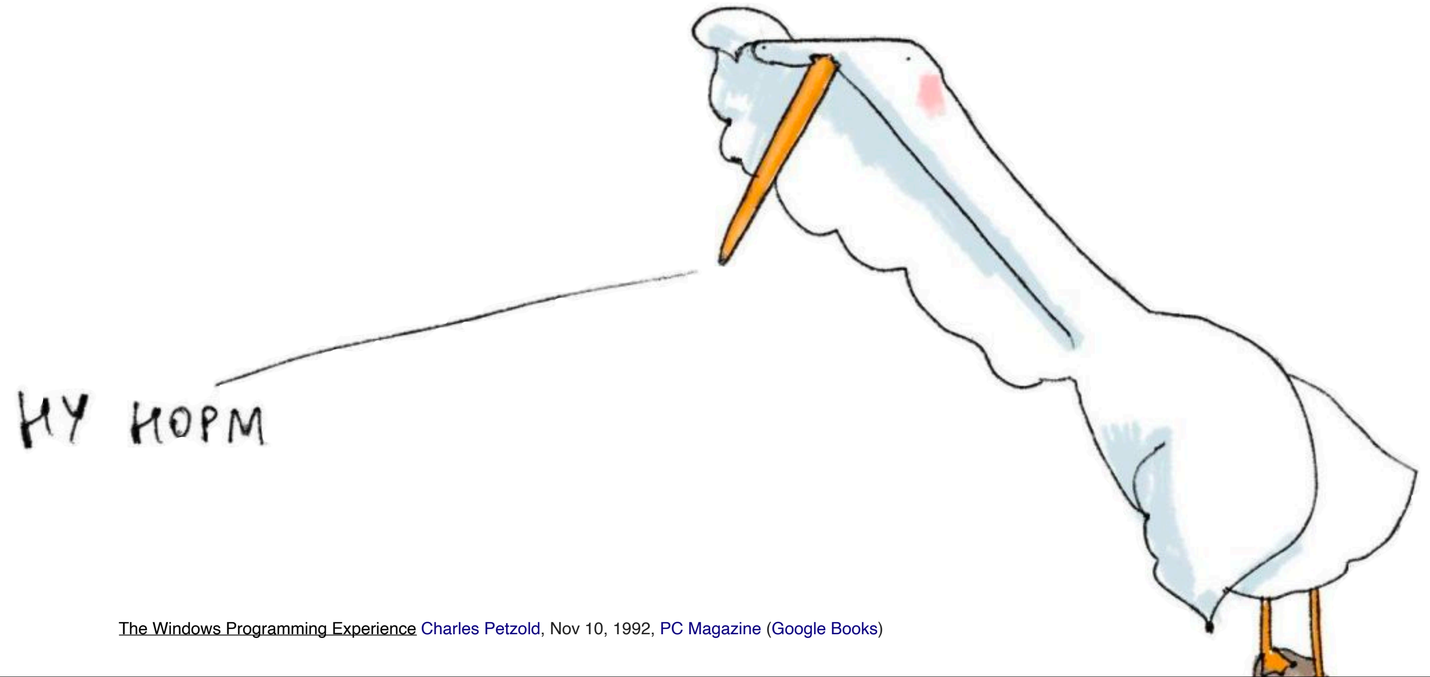
Windows programs use the GDI (Graphics Device Interface) for displaying text and graphics on the video display and printer. GDI supports both vector graphics and raster graphics. The same GDI function calls are used for both the video display and printers.

Over the past year, both Microsoft and Borland (as well as some other vendors) have released C++ tools for Windows programming. Microsoft's MFC (Microsoft Foundation Classes) and Borland's OWL (Object Windows Library) are application frameworks for Windows—prewritten code for common Windows programming tasks—that free the developer to focus more on the application logic. MFC and OWL differ in their layers of abstraction: MFC is largely a thin wrapping of the Windows API (application programming interface) with nearly 60 object classes, while OWL offers fewer classes but more effectively hides the API from developers.

Aspiring Windows programmers also now have available visual programming tools that provide a way to construct Windows applications without learning the Windows API (see our June 16, 1992, story "The Visual Development Environment: More than Just a Pretty Face?"). Borland's visual tool is ObjectVision; Microsoft offers Microsoft Visual BASIC. There are third-party products as well, such as Blue Sky Software's WindowsMaker, CaseWorks' Case:W, and ProtoView Development's ProtoGen. With these, you can create your application's interface with little more than a click of the mouse button.

Just as Windows has made it easier for users to work with applications, new tools have made it easier for programmers to create the applications. □

1992 (1995) - Паттерн Наблюдатель

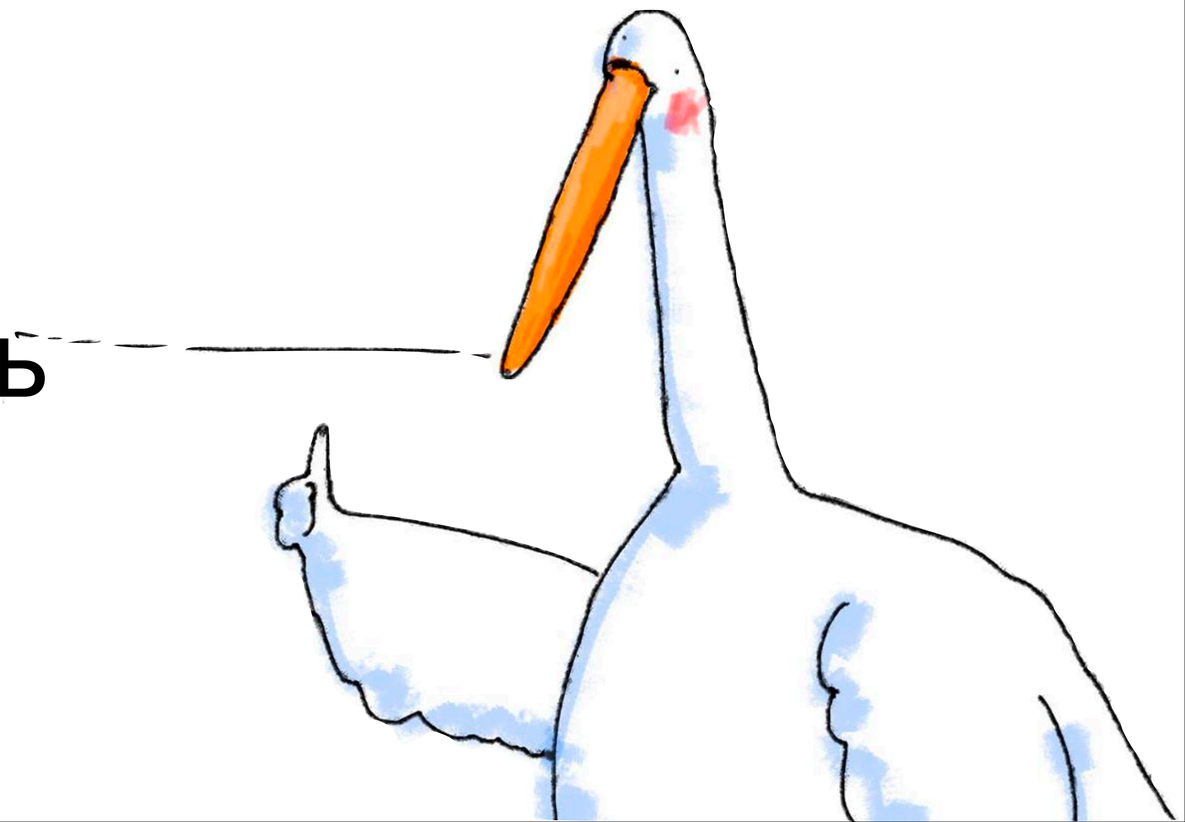


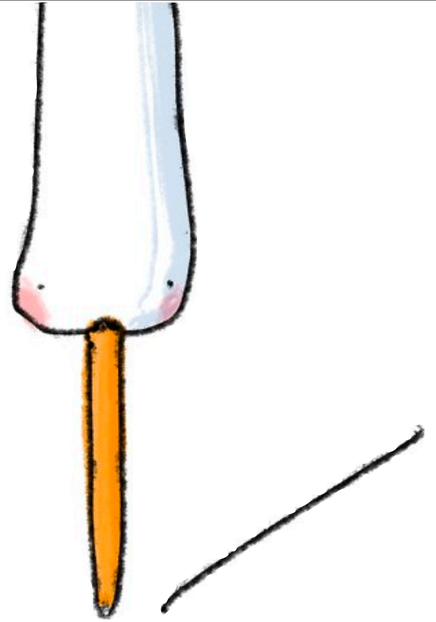
The Windows Programming Experience Charles Petzold, Nov 10, 1992, PC Magazine (Google Books)


Демка

```
const onMouseMoveObserver = (e: JQuery.Event) => {
    box.css({
        top: e.clientY - offsetY,
        left: e.clientX - offsetX
    })
}
box.mousemove(onMouseMoveObserver)
```

Агонь

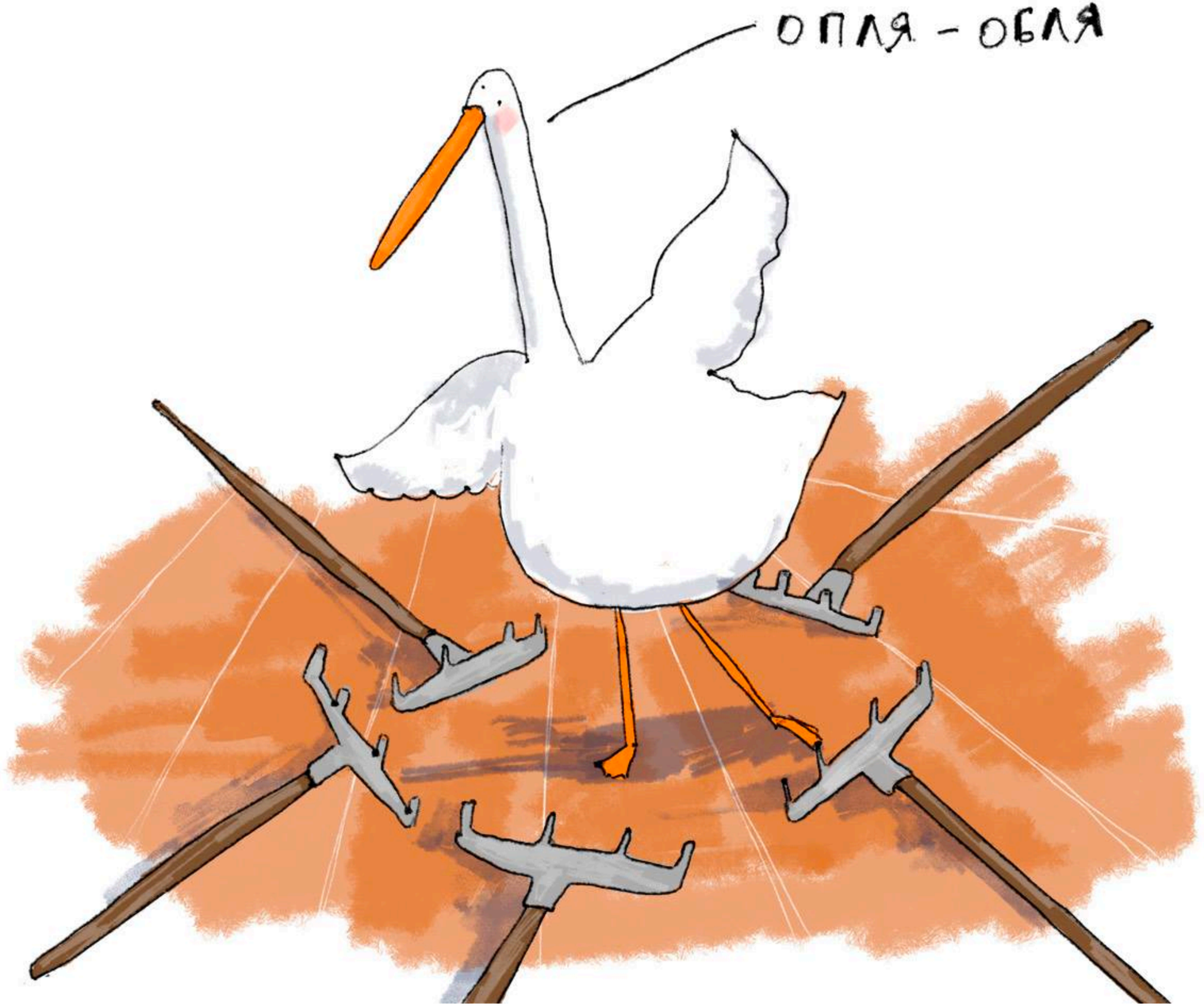




В

Все круто, но не совсем

На коромысле


ОПЛЯ - ОБЛЯ


Ой…
Я не хотел что бы так получилось

Проблематика

- Возникла необходимость построения сложной обработки событий
- Интерфейсы с не тривиальной анимацией
- Читаемость и прозрачность код стала снижаться

Functional Reactive Animation

Conal Elliott
Microsoft Research
Graphics Group
conal@microsoft.com

Paul Hudak
Yale University
Dept. of Computer Science
paul.hudak@yale.edu

Abstract

Fran (Functional Reactive Animation) is a collection of data types and functions for composing richly interactive, multimedia animations. The key ideas in Fran are its notions of *behaviors* and *events*. Behaviors are time-varying, reactive values, while events are sets of arbitrarily complex conditions, carrying possibly rich information. Most traditional values can be treated as behaviors, and when images are thus treated, they become animations. Although these notions are captured as data types rather than a programming language, we provide them with a denotational semantics, including a proper treatment of real time, to guide reasoning and implementation. A method to effectively and efficiently perform *event detection* using *interval analysis* is also described, which relies on the partial information structure on the domain of event times. Fran has been implemented in Hugs, yielding surprisingly good performance for an interpreter-based system. Several examples are given, including the ability to describe physical phenomena involving gravity, springs, velocity, acceleration, etc. using ordinary differential equations.

1 Introduction

The construction of richly interactive multimedia animations (involving audio, pictures, video, 2D and 3D graphics) has long been a complex and tedious job. Much of the difficulty, we believe, stems from the lack of sufficiently high-level abstractions, and in particular from the failure to clearly distinguish between *modeling* and *presentation*, or in other words, between *what* an animation is and *how* it should be presented. Consequently, the resulting programs must explicitly manage common implementation chores that have nothing to do with the content of an animation, but rather its presentation through low-level display libraries running on a sequential digital computer. These implementation chores include:

- stepping forward discretely in time for simulation and for frame generation, even though animation is conceptually continuous;
- capturing and handling sequences of motion input events, even though motion input is conceptually continuous;
- time slicing to update each time-varying animation parameter, even though these parameters conceptually vary in parallel; and

By allowing programmers to express the "what" of an interactive animation, one can hope to then automate the "how" of its presentation. With this point of view, it should not be surprising that a set of richly expressive recursive data types, combined with a declarative programming language, serves comfortably for modeling animations, in contrast with the common practice of using imperative languages to program in the conventional hybrid modeling/-presentation style. Moreover, we have found that non-strict semantics, higher-order functions, strong polymorphic typing, and systematic overloading are valuable language properties for supporting modeled animations. For these reasons, Fran provides these data types in the programming language Haskell [9].

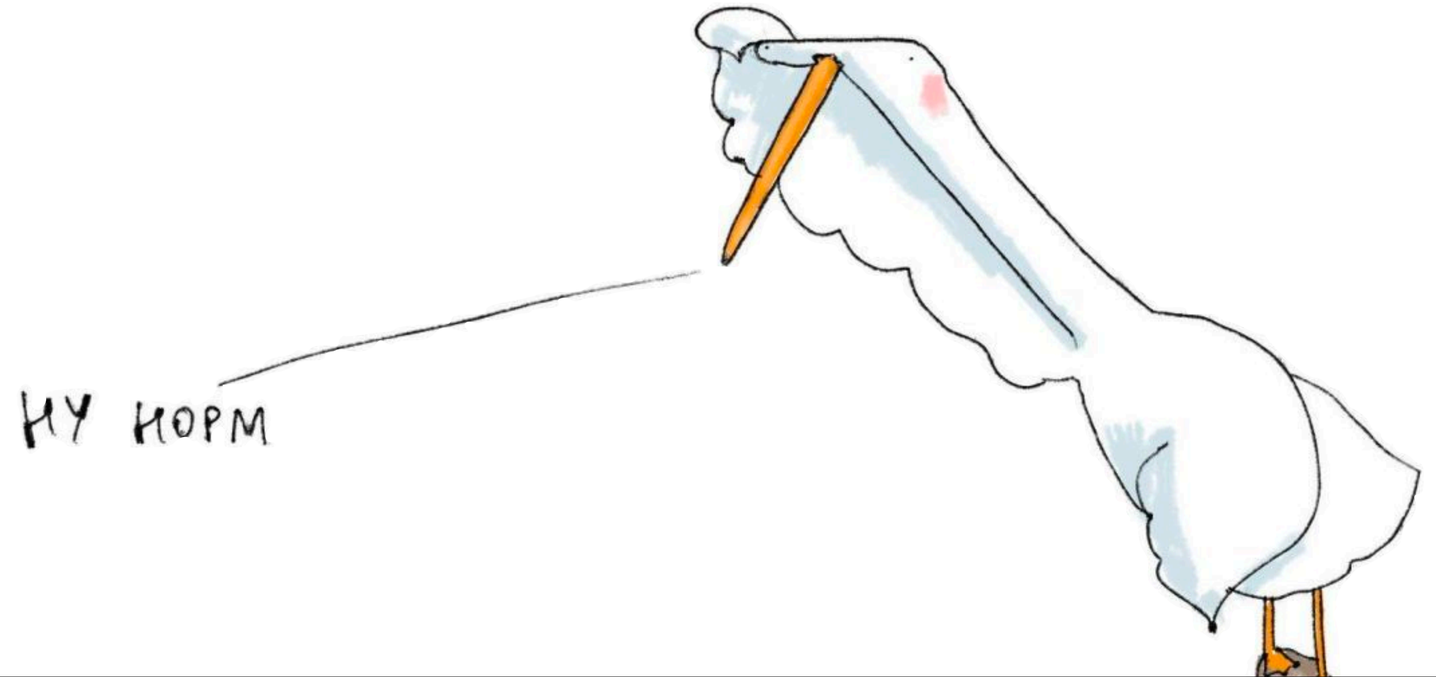
Advantages of Modeling over Presentation

The benefits of a modeling approach to animation are similar to those in favor of a functional (or other declarative) programming paradigm, and include clarity, ease of construction, composability, and clean semantics. But in addition there are application-specific advantages that are in some ways more compelling, painting the picture from a software engineering and end-user perspective. These advantages include the following:

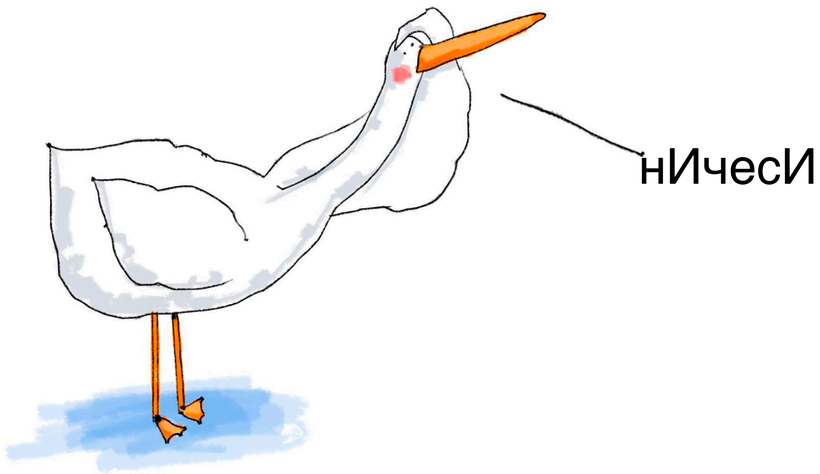
- *Authoring.* Content creation systems naturally construct models, because the end users of such systems think in terms of models and typically have neither the expertise nor interest in programming presentation details.
- *Optimizability.* Model-based systems contain a presentation sub-system able to render any model that can be constructed within the system. Because higher-level information is available to the presentation sub-system than with presentation programs, there are many more opportunities for optimization.
- *Regulation.* The presentation sub-system can also more easily determine level-of-detail management, as well as sampling rates required for interactive animations, based on scene complexity, machine speed and load, etc.

To appear the International Conference on Functional Programming, June 1997, Amsterdam.

1997 - Functional Reactive Animation



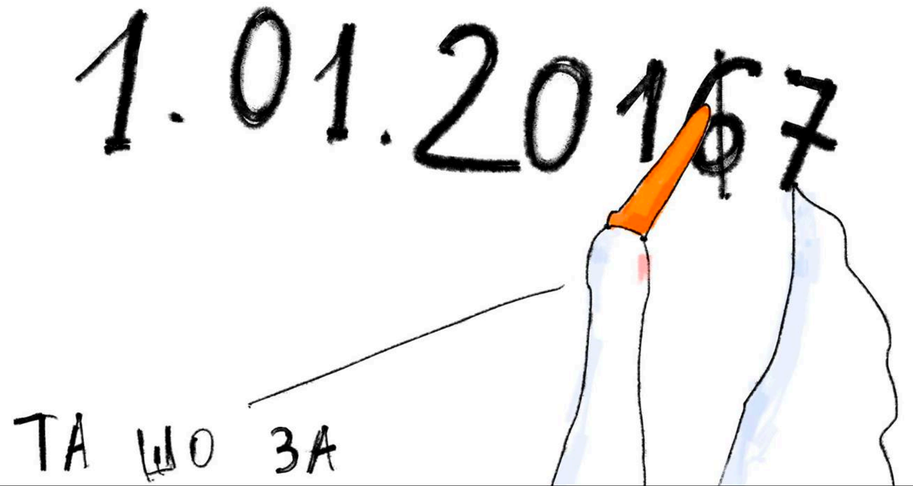
```
kids u =
  delayAnims 0.5
    (map (move (mouseMotion u))
         [jake, becky, charlotte, pat])
```



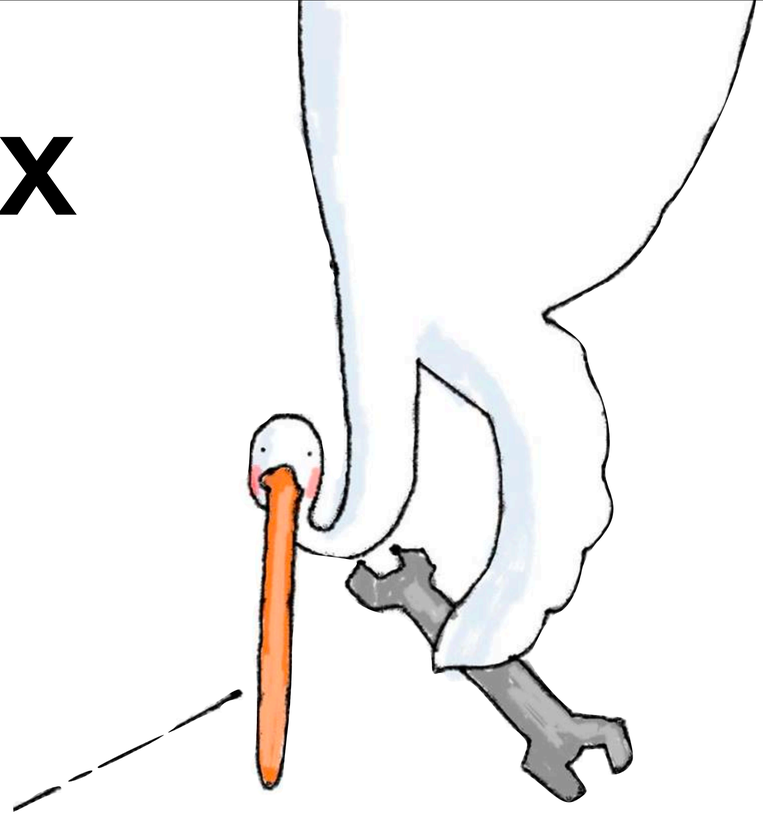
Итог - все круто

- Паттерн Наблюдатель как подход
- Functional Reactive Animation как языковая парадигма

Все круто, но не совсем



Проблематика 00-ых



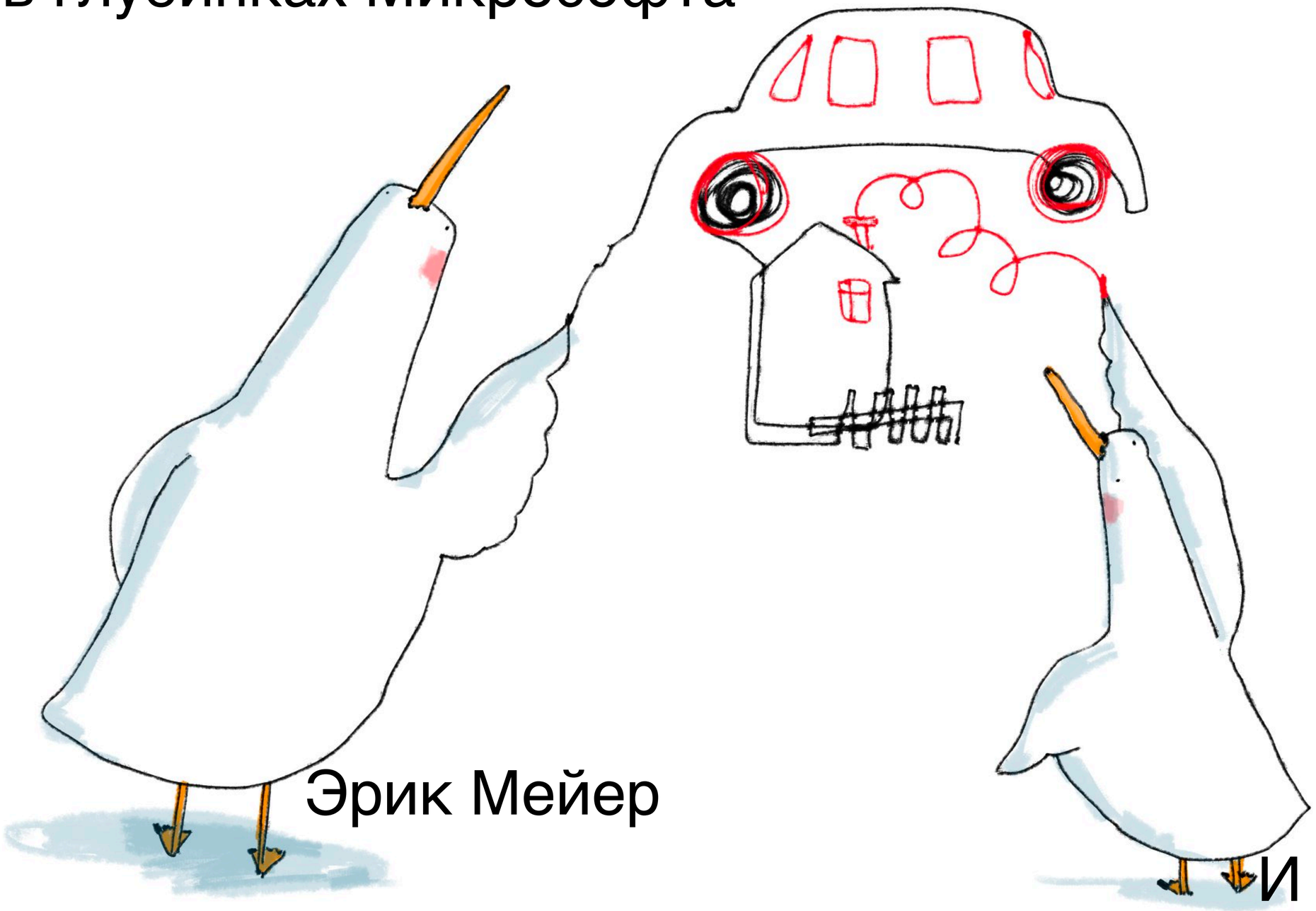
Для построения эффективного приложения необходимы соответсвующие подходы


Демка

Недостатки подходов

- Ходки за одним **НЕ** эффективны из за **задержки** сети
- **Пропускная** способность сети **НЕ** безгранична
- **Ресурсы** компьютера **ограничены**

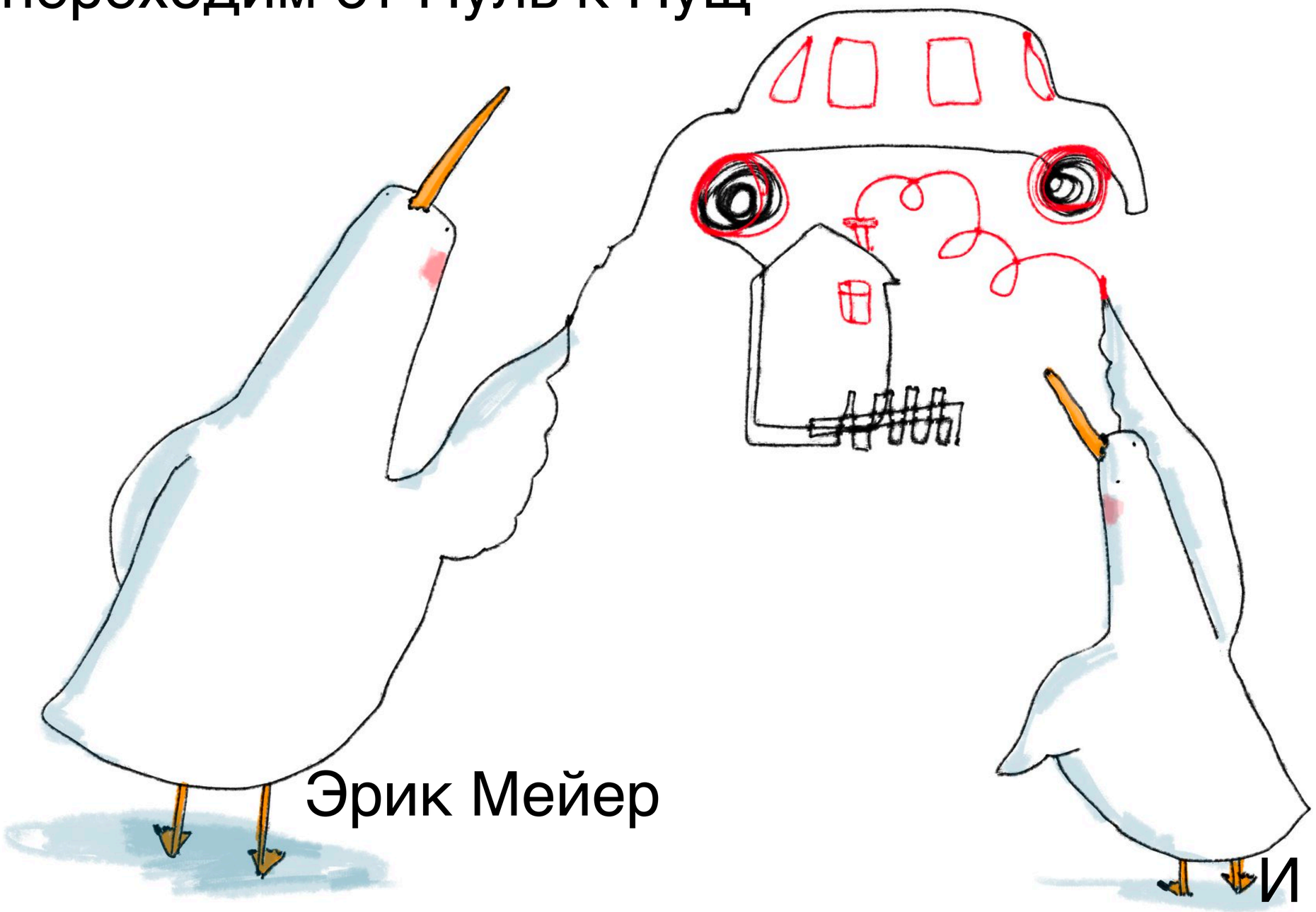
...где то в глубинках Микрософта



Эрик Мейер

И команда

Эврика - переходим от Пуль к Пущ



Эрик Мейер

И команда


Демка


Короче…
как так
получилось…

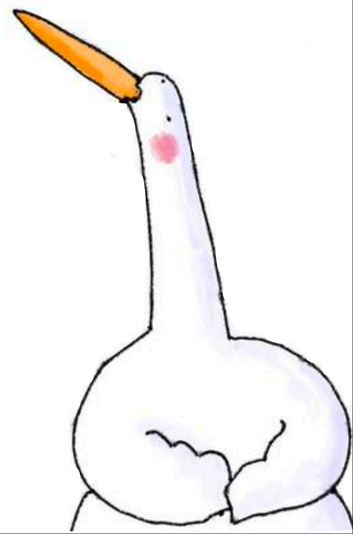
Проблематика

Observer

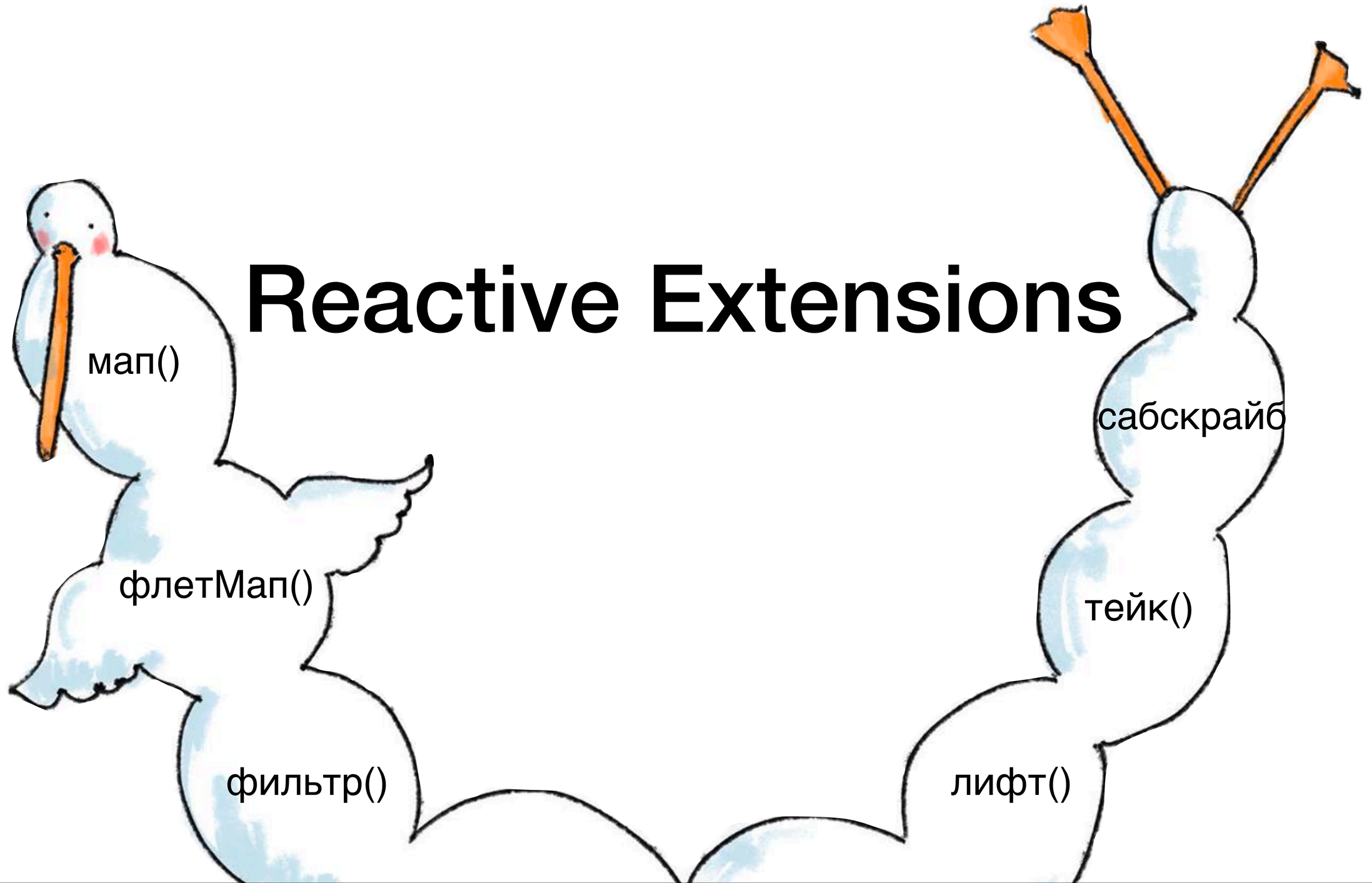
+ Асинхронный Push
- А где конец?
- А как отписаться?

Iterator

- Синхронный Pull
+ Ясно, где конец
+ Можем отписаться



Reactive Extensions



В чем суть

Observer + Iterator = Rx

Observable

{ subscribe() }

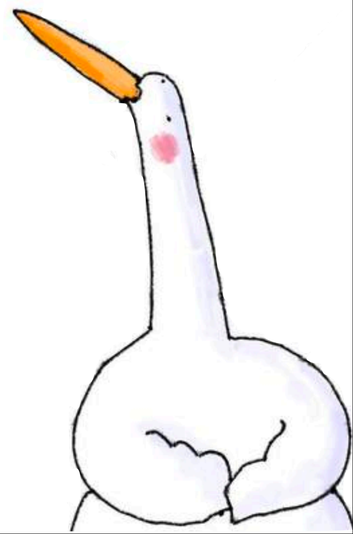
Subscription

{ cancel() }

Observer

{ onNext
onError
onComplete }

+ Асинхронный Push
+ Ясно, где конец
+ Понятно, как отписаться

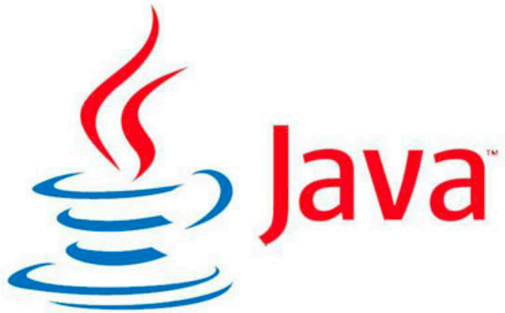



ДСЛ

Языковая парадигма

```
range(1, 200)
.pipe(
  filter(x => x % 2 === 1),
  map(x => x + x)
)
.subscribe(x => console.log(x));
```


```
Observable
  .range(1, 200)
  .filter(x -> x % 2 === 1)
  .map(x -> x + x)
  .subscribe(x -> out.println(x));
```



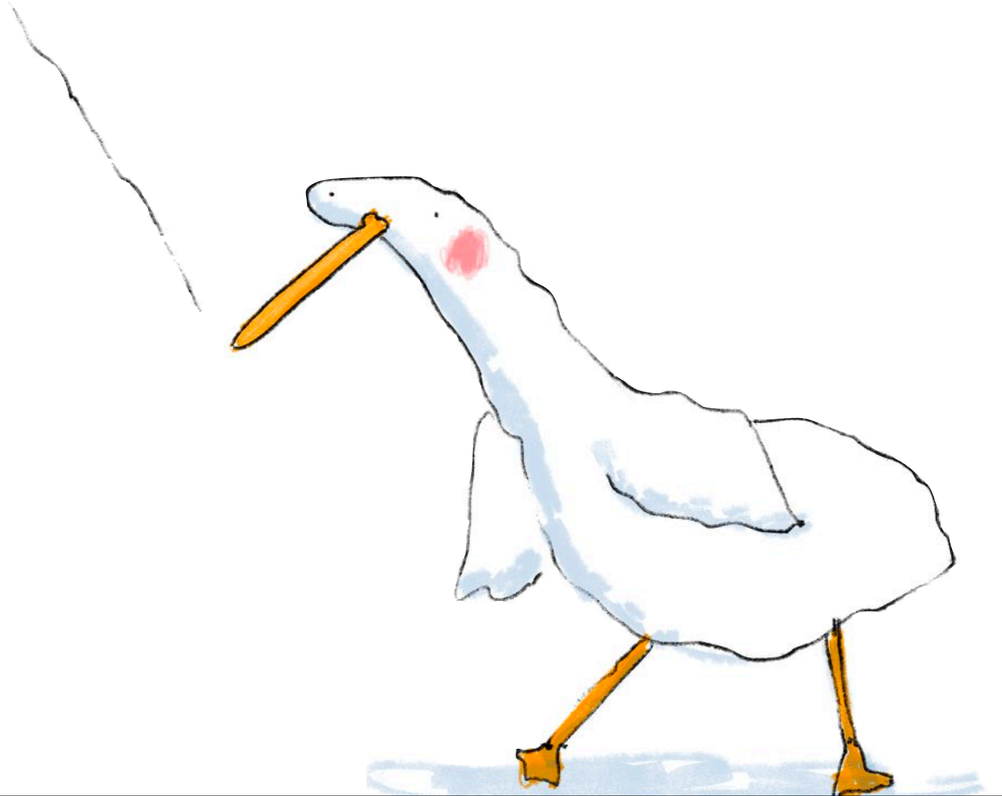

```
Observable
  .range(1, 200)
  .filter{ $0 % 2 === 1 }
  .map{ $0 + $0 }
  .subscribe(onNext: { print($0) });
```

Итог - все круто

- Rx подход решил проблемы
- Построение сложных асинхронных взаимодействий вновь стало простым

Все круто, но не совсем

Неужели опять




Демка

Проблематика

- Один языковой подход но разное поведение
- Медленный подписчика и быстрый производитель

Reactive Streams Specification

В чем суть

Rx + Batch = RS

Push

Pull

Publisher

{ subscribe() }

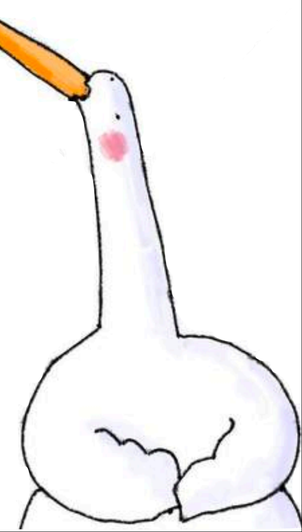
Subscriber

{ onSubscribe
onNext
onError
onComplete }

Subscription

{ cancel()
request() }

+ Rx Push модель
+ Async Batch pull
+ Стандартный набор интерфейсов




Демка

Наши дни

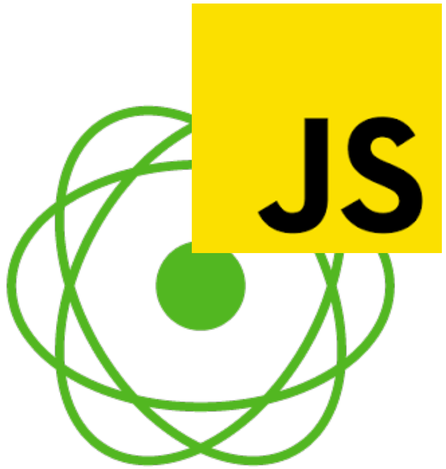
Adoption

Библиотеки

Reactive4NET

Project Reactor

RxJava



ReactorJS

А в JVM вообще огонь

JEP 266: More Concurrency Updates

Owner	Doug Lea
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	9
Component	core-libs / java.util.concurrent
Discussion	core dash libs dash dev at openjdk dot java dot net
Effort	S
Duration	S
Relates to	JEP 193: Variable Handles
Reviewed by	Chris Hegarty, Martin Buchholz, Paul Sandoz
Endorsed by	Brian Goetz
Created	2015/08/04 11:01
Updated	2017/04/24 23:02
Issue	8132960

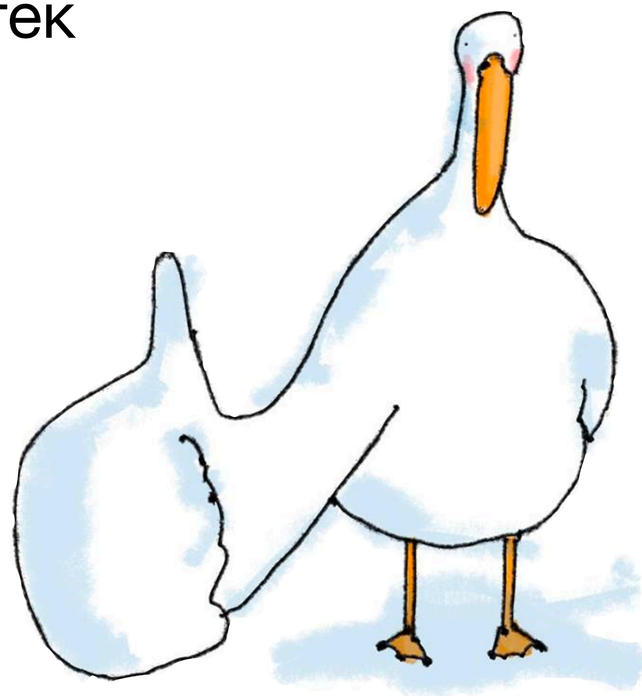
Summary

An interoperable publish-subscribe framework, enhancements to the `CompletableFuture` API, and various other improvements.

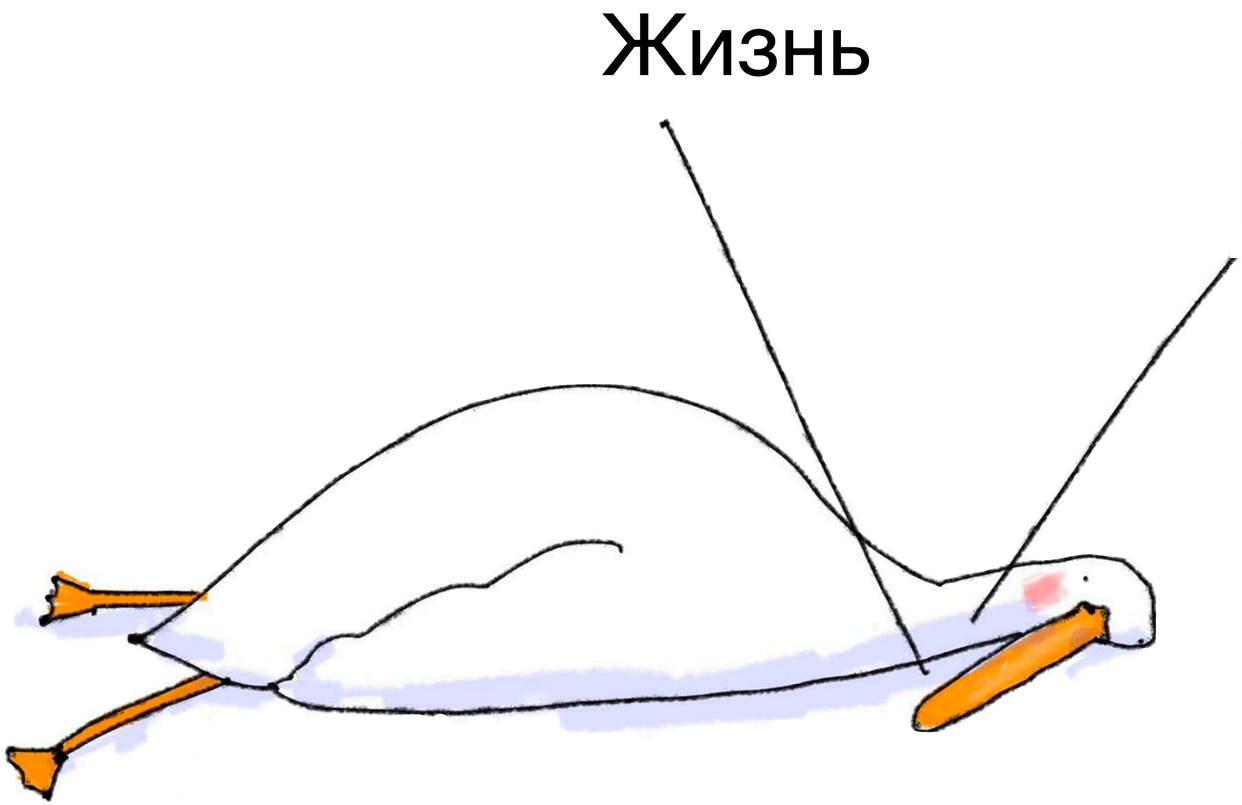
https://openjdk.java.net/jeps/266

Итог - все круто

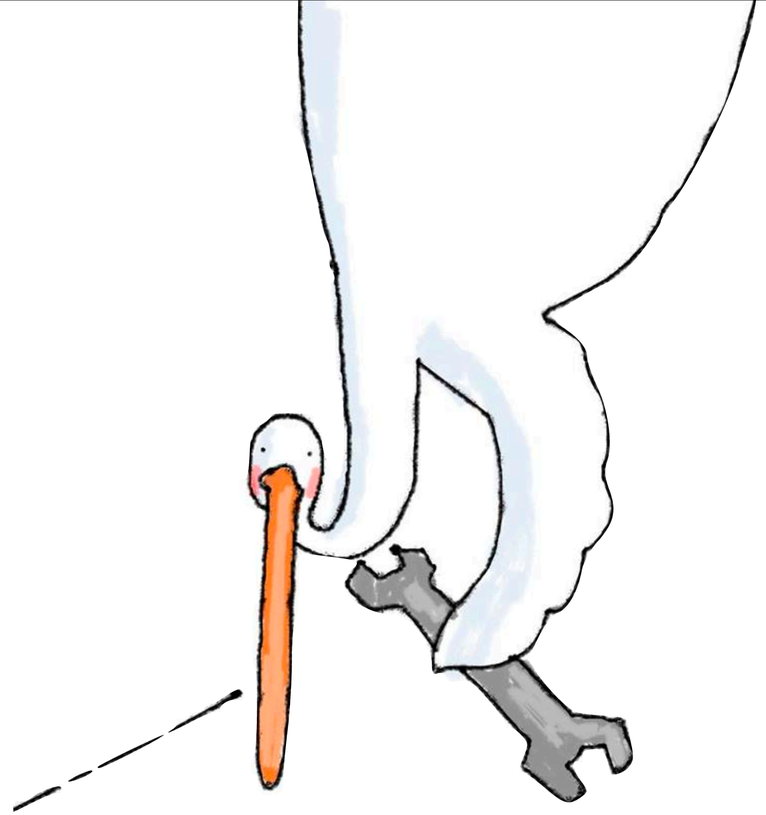
- Reactive-Streams spec - стандарт реактивных библиотек
- Улучшеный API с Push-Pull стримингом



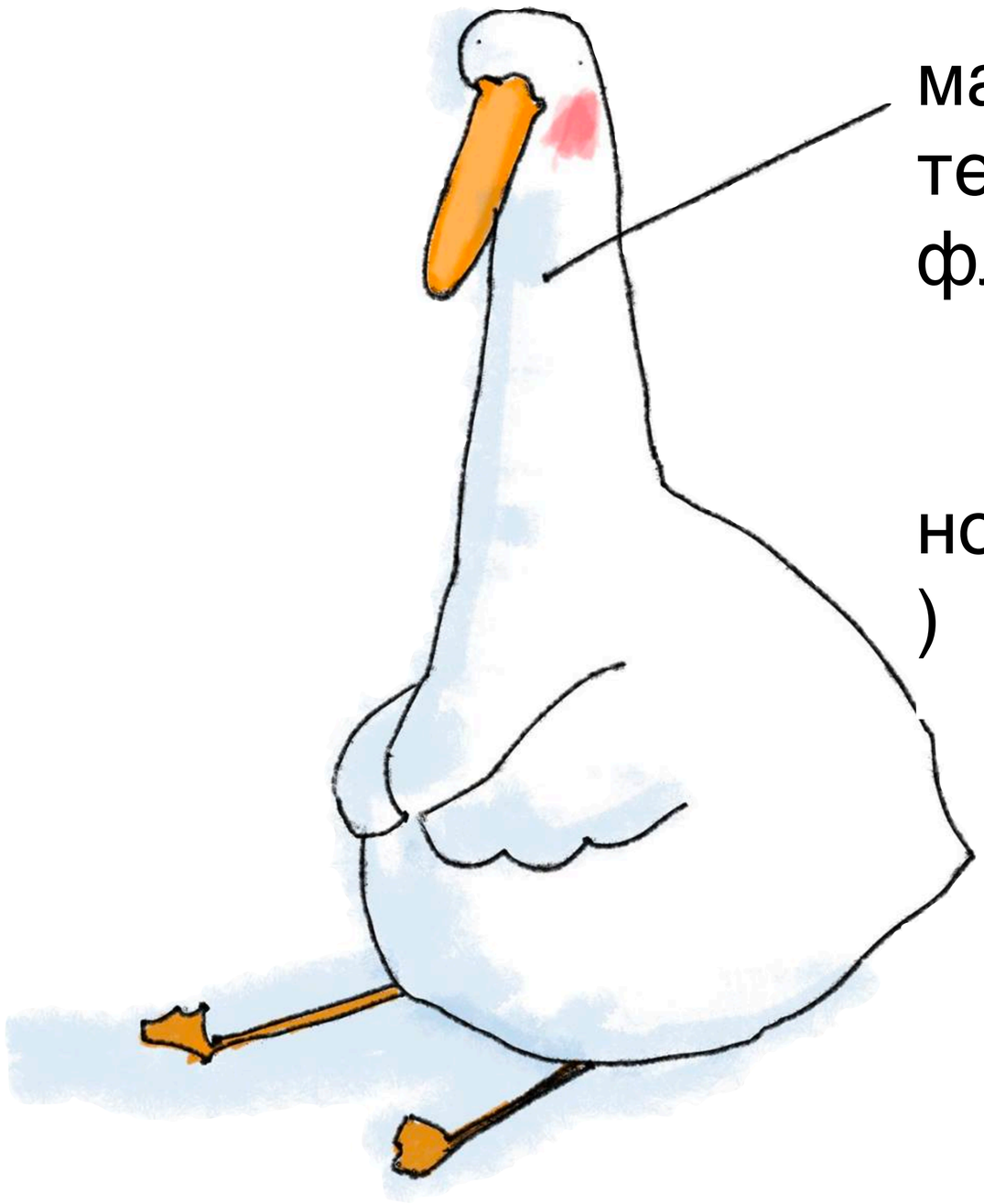
Все круто, но опять…



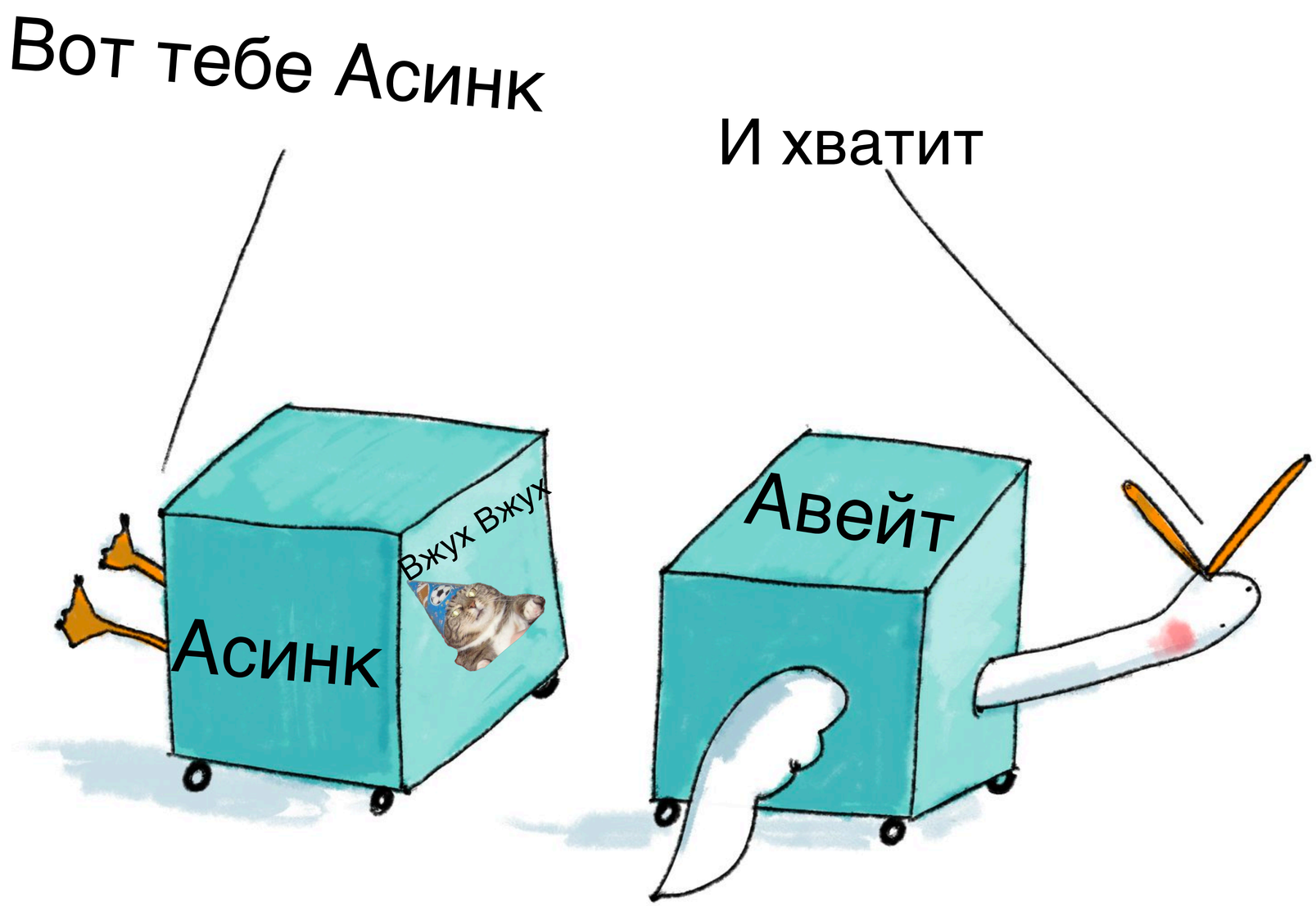
Проблематика



Все круто но… я программист и я не хочу решать проблемы



```
мап(() ->{ … })
тейк(3)
флетМап(() -> …
    флетМаП(() ->
        И два выстрела в
ногу())
)
```


Вот тебе Асинк
И хватит
Асинк
Вжух Вжух
Авейт

Answered by:

2,445
Points
Top 5%

Jeffrey van Gogh - MS

Joined Nov 2005

Jeffrey van Gog...

1 3 9

Show activity

Is IObservable dead?

Data Platform Development > Reactive Extensions (Rx)

Question

Watching Anders demoing async/await - looks like it is a replacement for RX?

Thursday, October 28, 2010 6:57 PM

Reply | Quote

Sergey Aldoukhov 140 Points

0

Sign in to vote

Answers

6

Sign in to vote

Hi Sergey & Ray,

Rx is definitely not dead :). The Rx team has worked closely with the C# and VB teams to develop the await/async functionality. This new pattern will make it easier for developers to write asynchronous sequential code. For this they can either use Task or IObservable (see today's Rx release notes).

As part of our release today, we have also introduced an asynchronous sequential stream (IAsyncEnumerable). This allows developers to easily use the await keyword to deal with pull based streams that are asynchronous.

However, there are other streams that are inherently parallel. With these streams it is still not

Async Streams

02/25/2019 • 20 minutes to read •  

- [x] Proposed
- [x] Prototype
- [ ] Implementation
- [ ] Specification

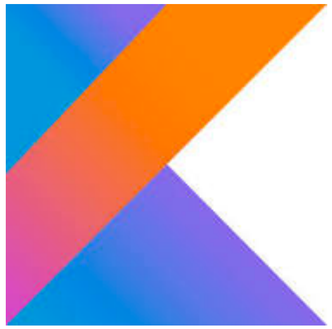
Summary

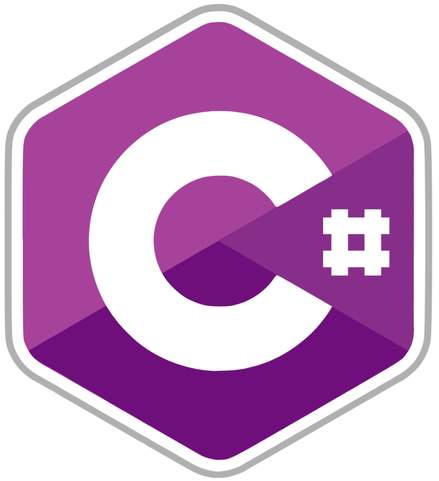
C# has support for iterator methods and async methods, but no support for a method that is both an iterator and an async method. We should rectify this by allowing for `await` to be used in a new form of `async` iterator, one that returns an `IAsyncEnumerable<T>` or `IAsyncEnumerator<T>` rather than an `IEnumerable<T>` or `IEnumerator<T>`, with `IAsyncEnumerable<T>` consumable in a new `await foreach`. An `IAsyncDisposable` interface is also used to enable asynchronous cleanup.

В чем суть

```
IAsyncEnumerator<T> enumerator = enumerable.GetAsyncEnumerator();
try
{
    while (await enumerator.WaitForNextAsync())
    {
        while (true)
        {
            int item = enumerator.TryGetNext(out bool success);
            if (!success) break;
            Use(item);
        }
    }
}
finally { await enumerator.DisposeAsync(); }
```

Async / Await в языковом мире





OpenJDK

Workshop

OpenJDK FAQ
Installing
Contributing
Sponsoring
Developers' Guide
Vulnerabilities

Mailing lists
IRC · Wiki

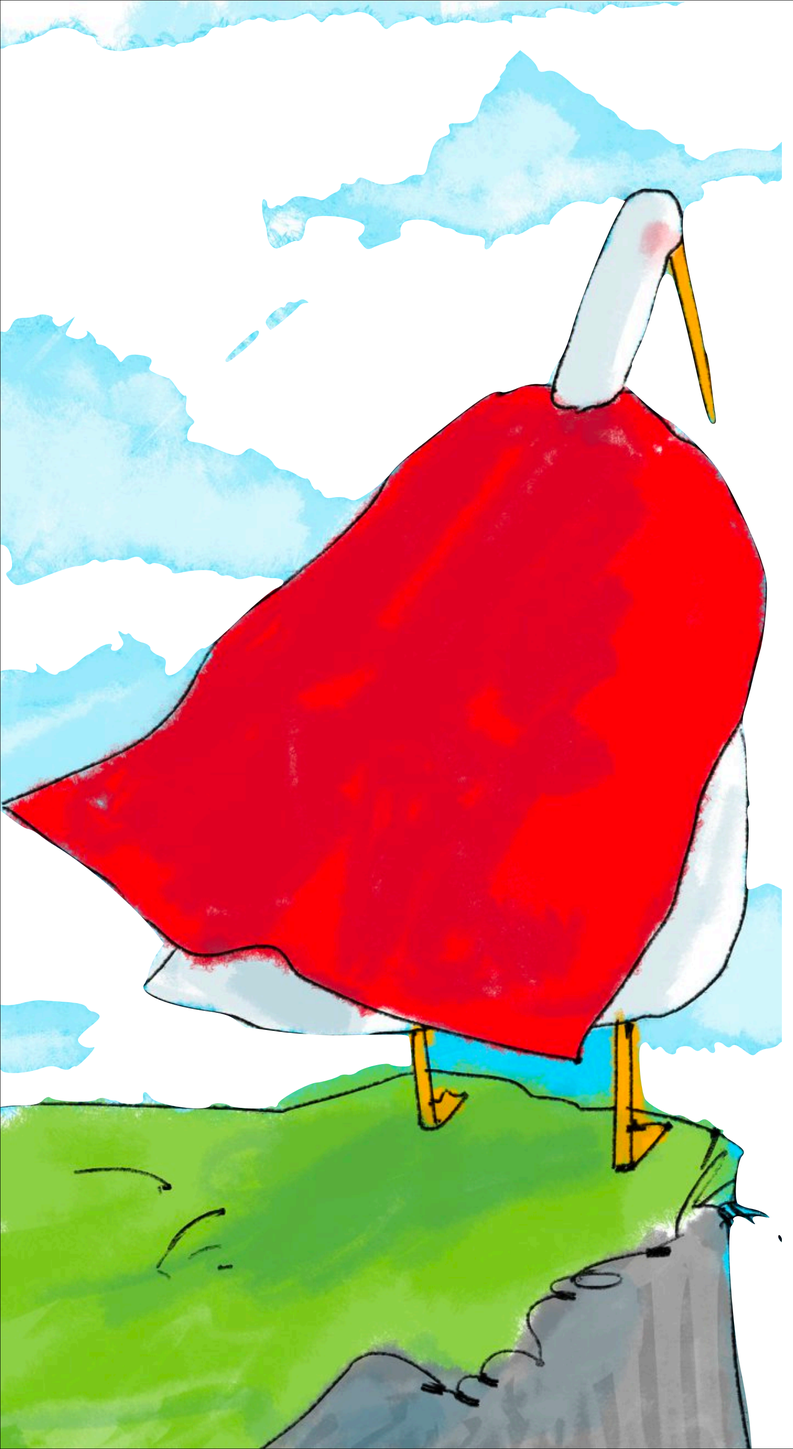
Bylaws · Census

Loom - Fibers, Continuations and Tail-Calls for the JVM

PLEASE NOTE! Go to the Wiki for additional and up-to-date information.

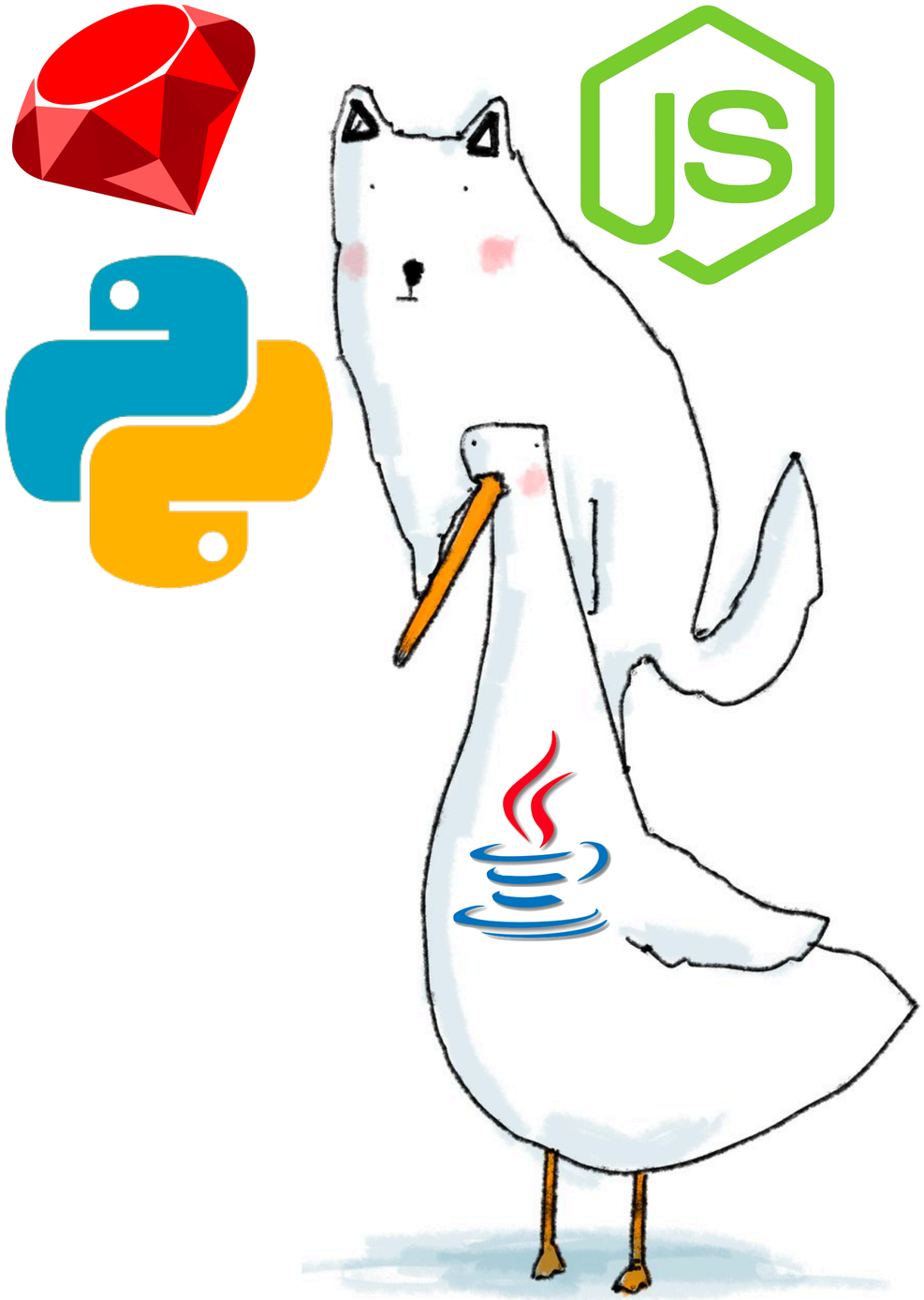
The goal of this Project is to explore and incubate Java VM features and APIs built on top of them for the implementation of lightweight user-mode threads (fibers), delimited continuations (of some form), and related features, such as explicit tail-call.

This Project is sponsored by the HotSpot Group.

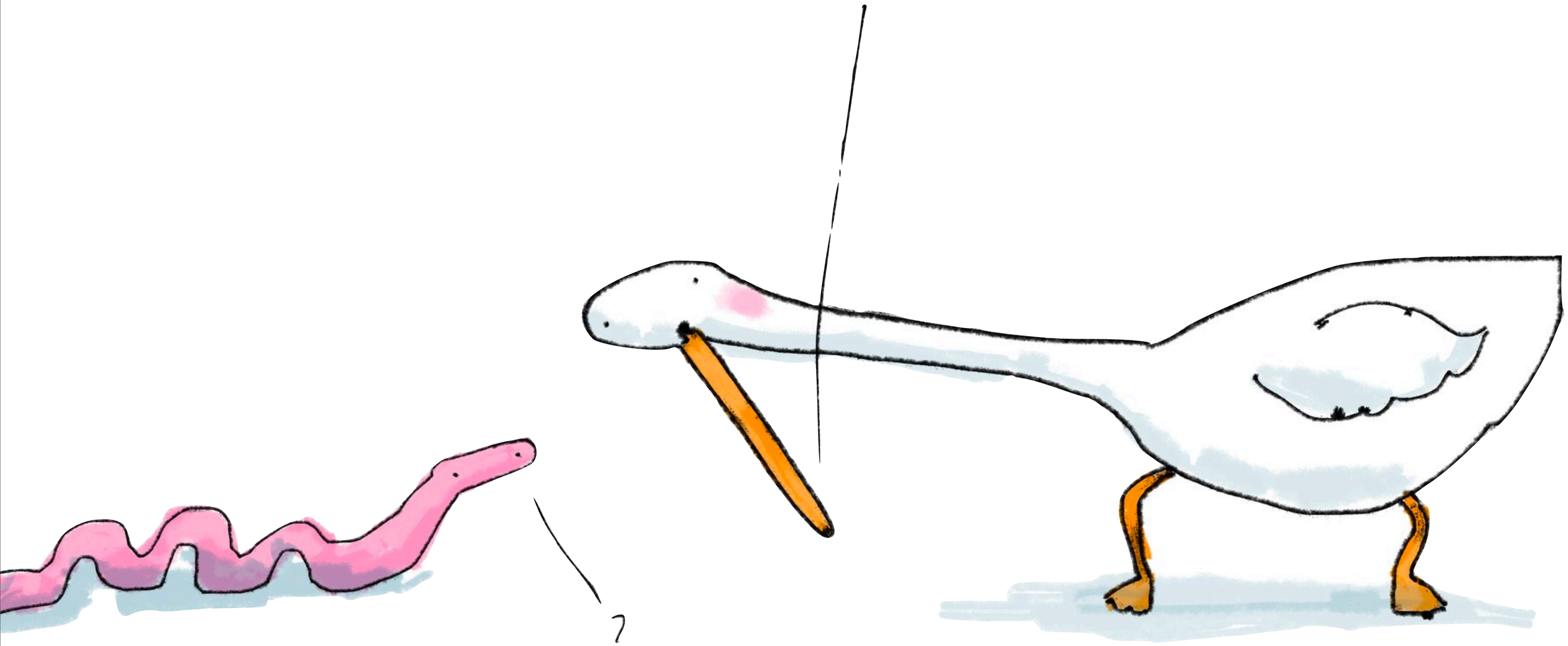


Так что же нас ждет?

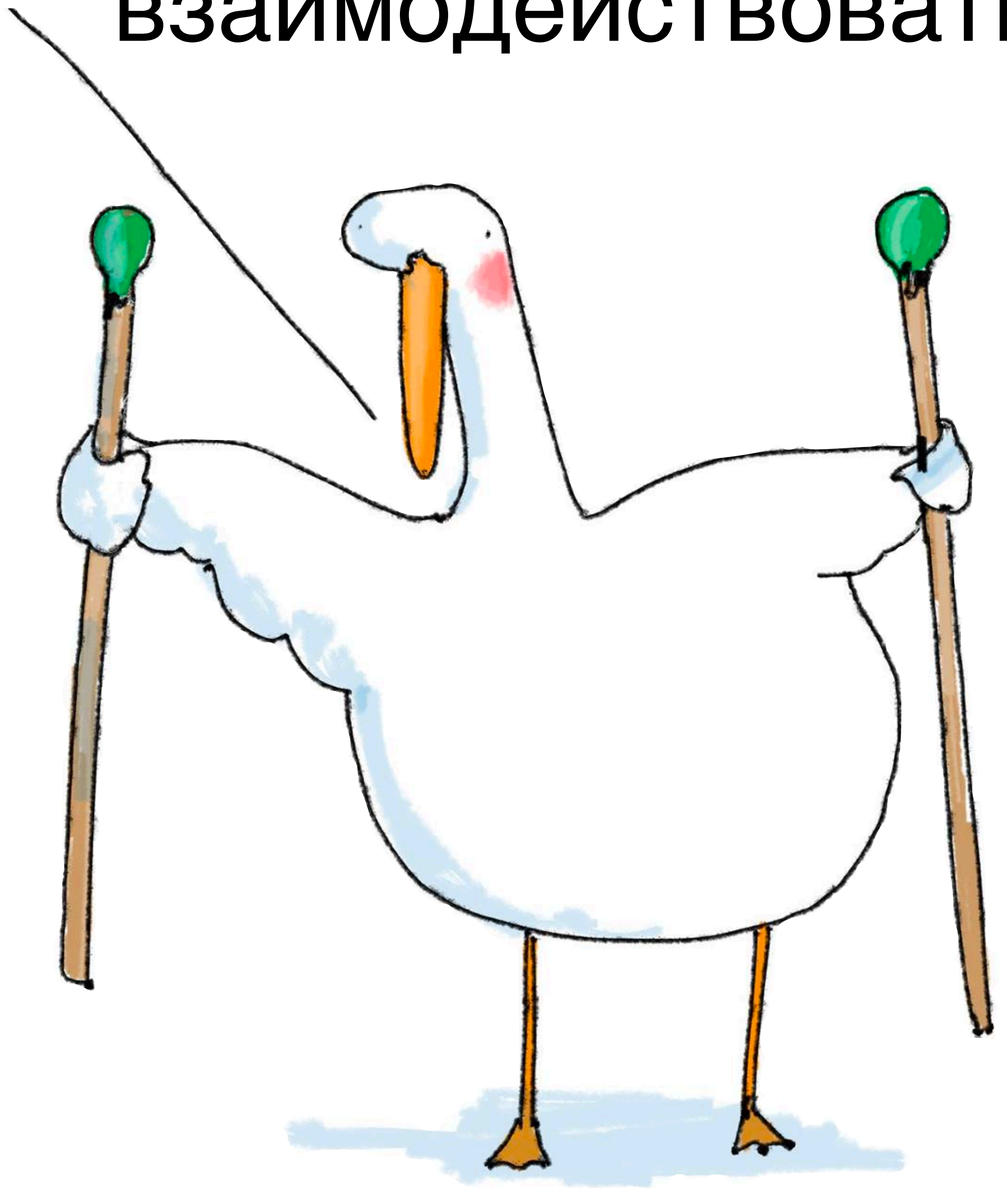

GraalVM™


JS

Тут-то и возникает вопросик



А как эти ваши Асинк/Авейты будут взаимодействовать?



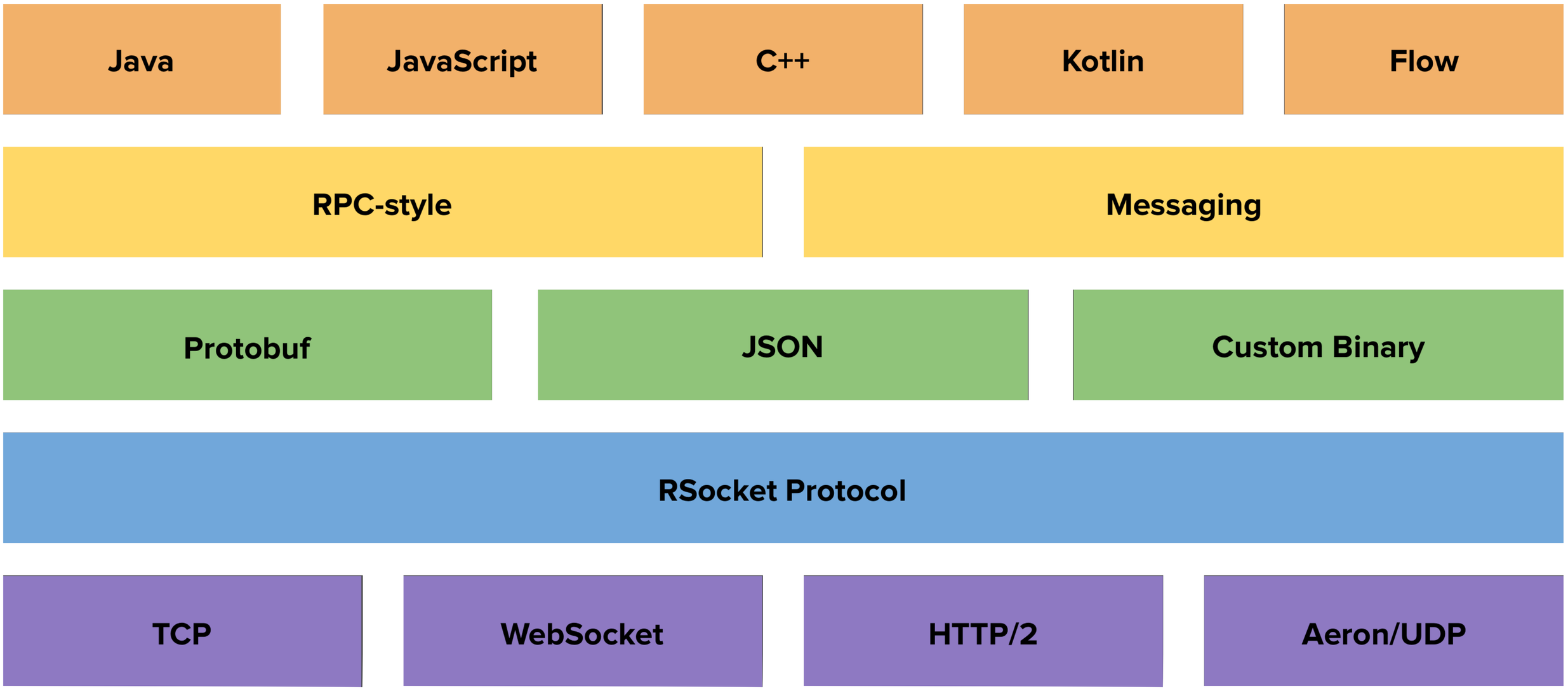
А стандарт есть стандарт

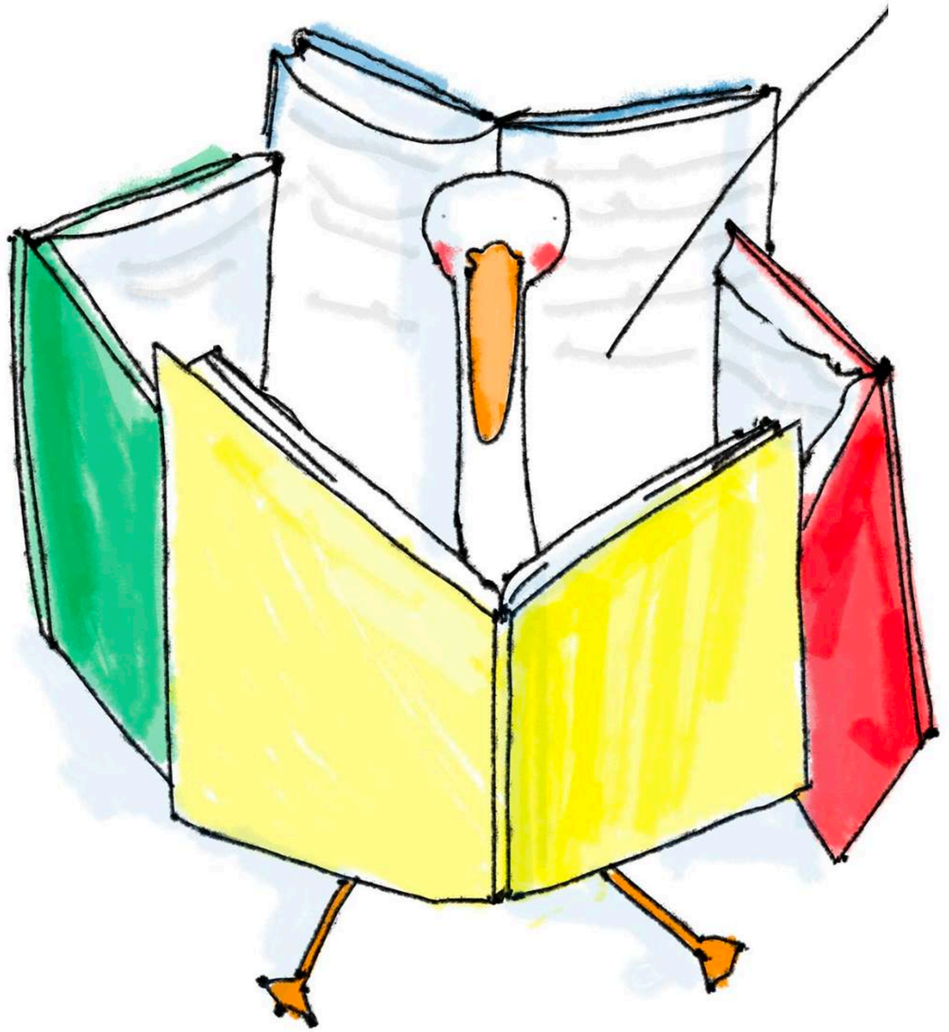
Сила в спецификации!!!


Демка

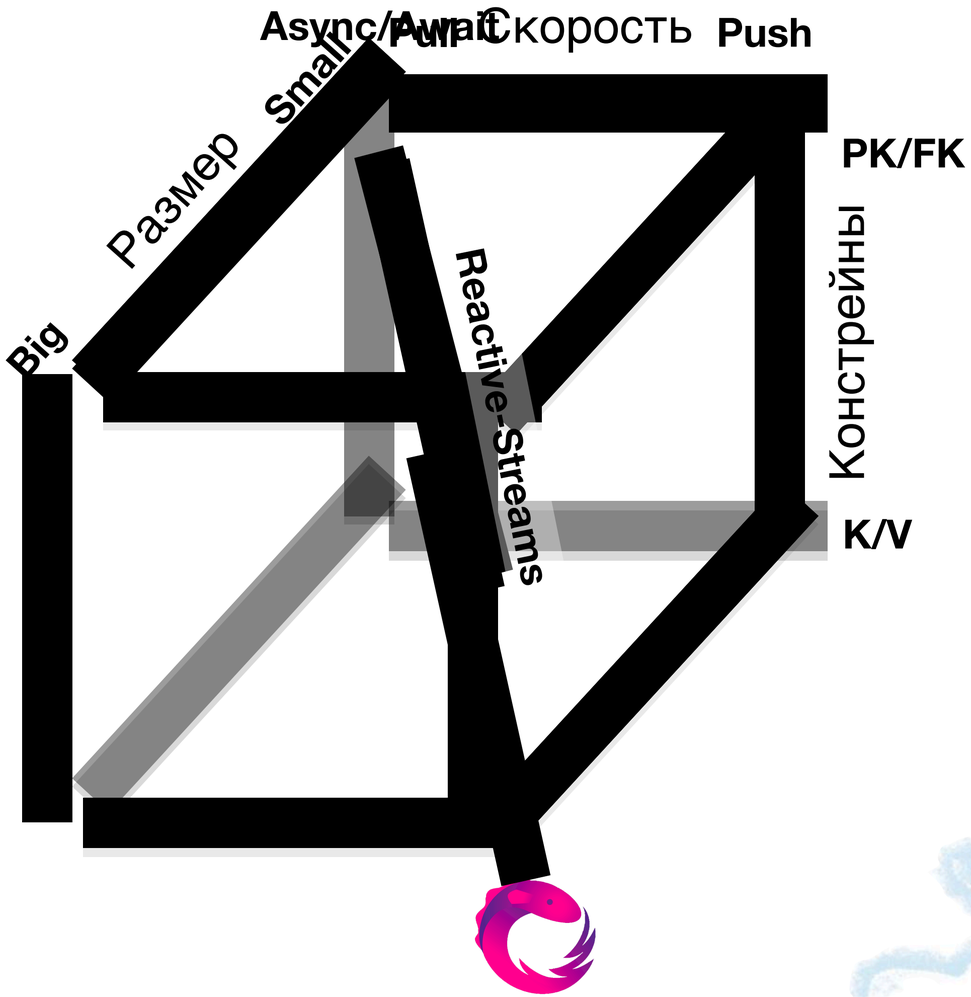

Socket

Reactive Streams as Protocol



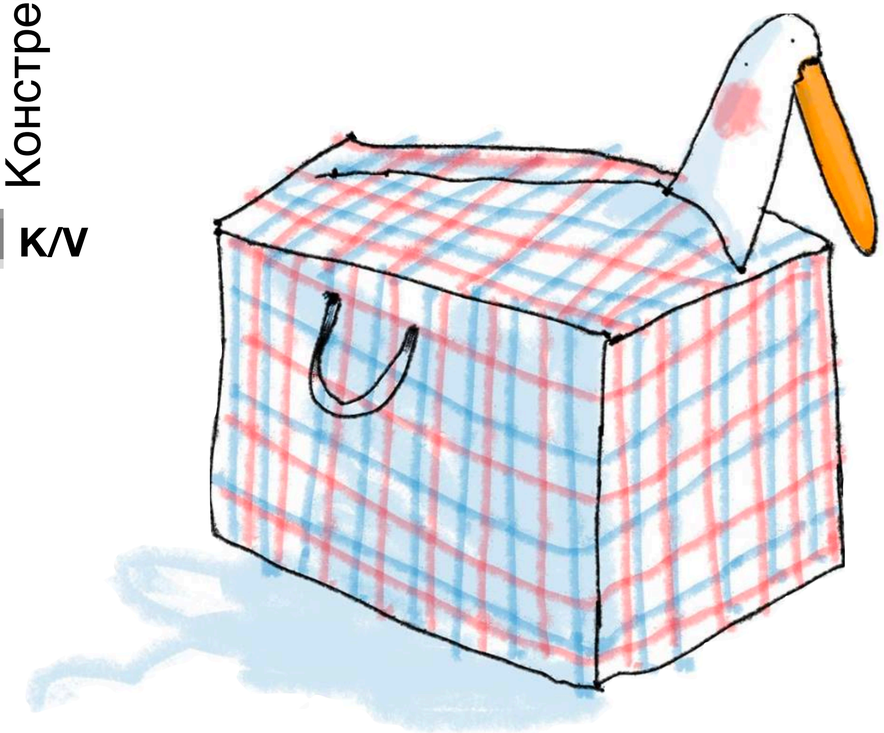


Так есть ли будущее у Реактивных Стримов



Куб от Гуся…

шутка -
от Эрика Мейера

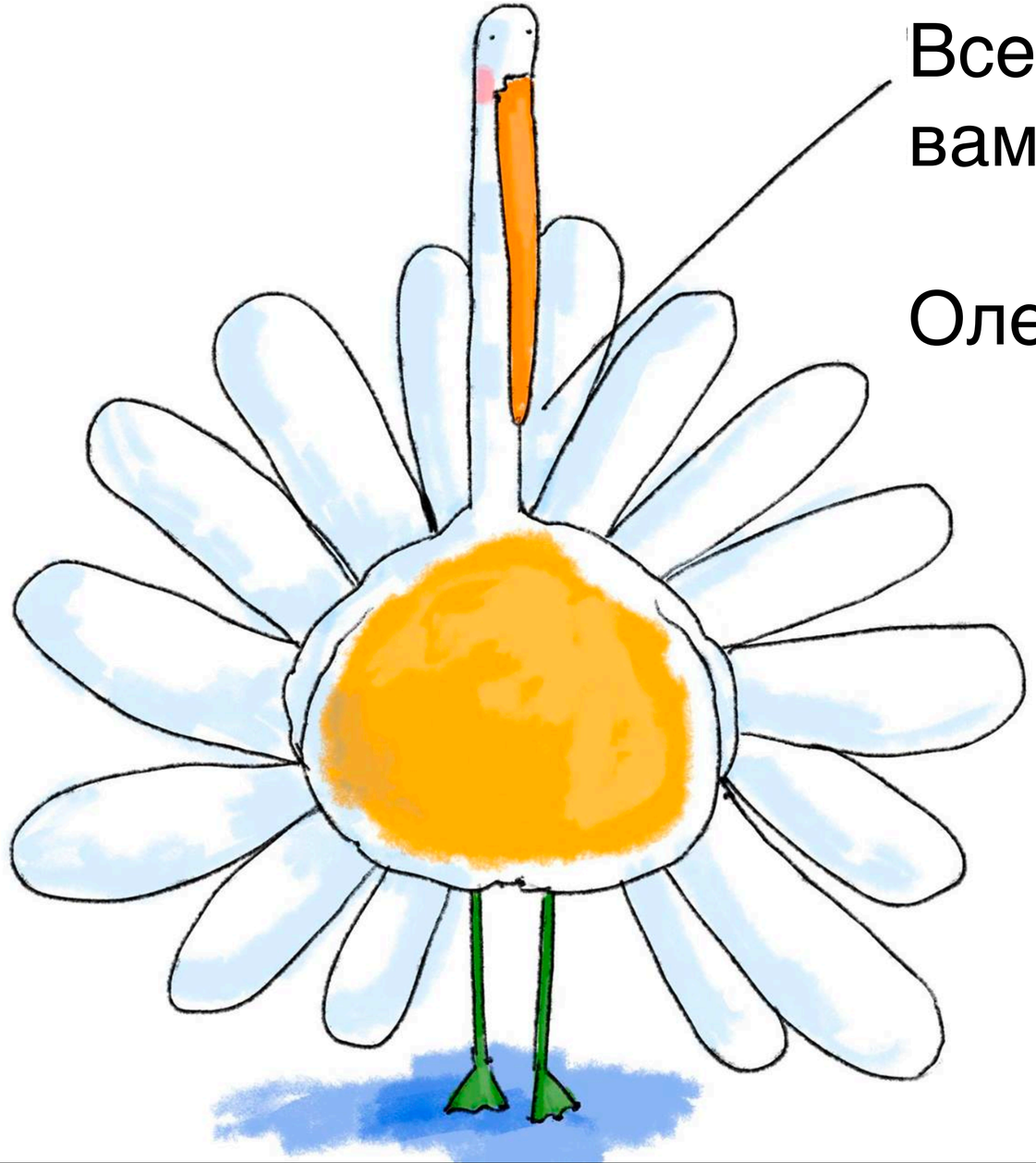


https://queue.acm.org/detail.cfm?id=2169076

Финальный Итог

- Эволюция Reactive Streams начала с давных времен
- Зародилась в мире UI и нашла свою жизнь в мире EE приложений
- Все это выродилось в стандарт нацеленный на стабильность системы
- Стандарт есть стандарт - работает везде и также
- Используй все это с умом подглядывая на куб от гуся!

https://www.facebook.com/thegooseishere/



Всем СпасибА, с вами был Гу…

Олег