UNIVERSITY OF WATERLOO
FACULTY OF MATHEMATICS

Declarative Specifications for Software Code Base

Software Engineering using the functional paradigm

Contents

01 **The end of imperative**

Assumptions, preventions, problems

02 **The functional approach**

Currying, purity, state explosion

03 **The lambda architecture**

Breaking the dependencies, splitting the truth

04 **Where to start?**

Tutorials, technologies, websites

The key concept

Assist REs with greater flexibility and easier proof in the code.

Imperative Vs Declarative

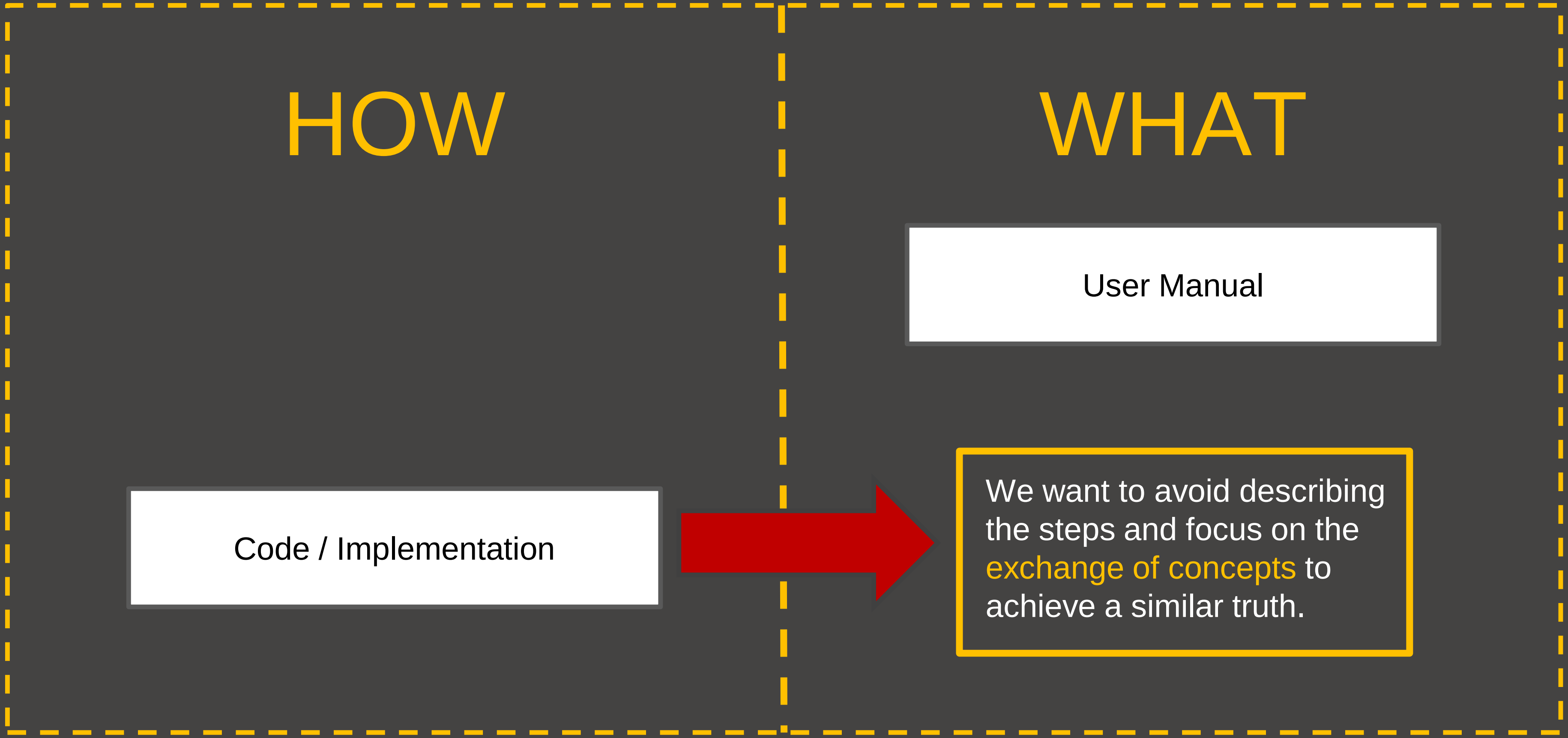
HOW

Code / Implementation

WHAT

User Manual

Imperative Vs Declarative



HOW

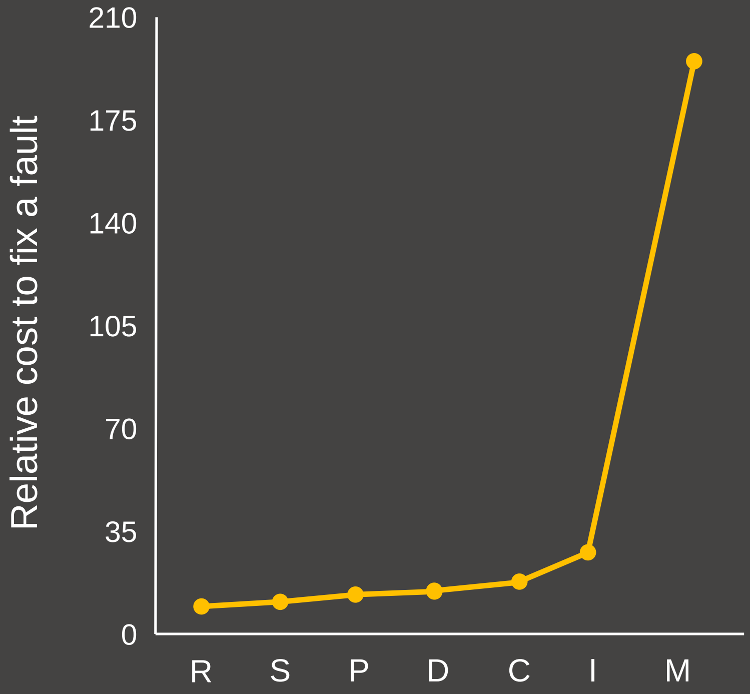
WHAT

User Manual

Code / Implementation

We want to avoid describing the steps and focus on the exchange of concepts to achieve a similar truth.

Engineering cycle



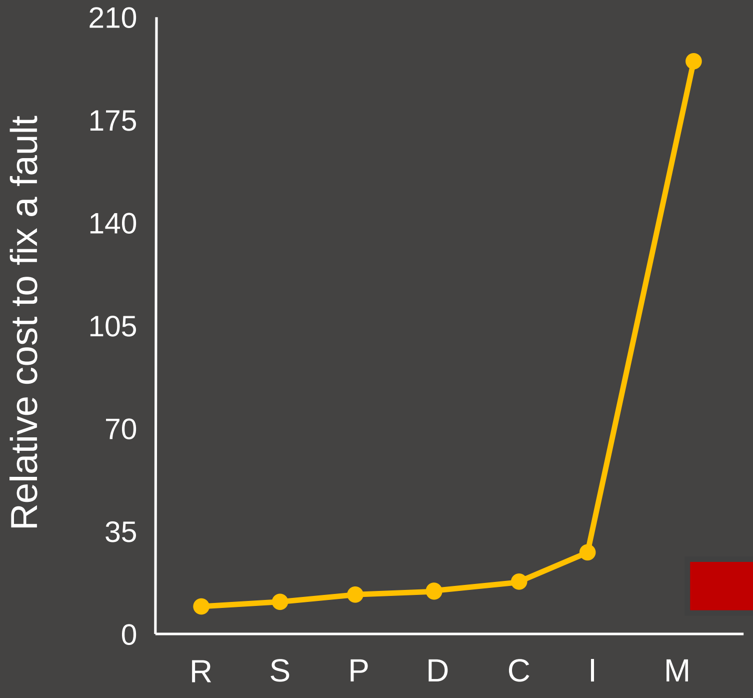
Requirements
Specifications
Plan
Design

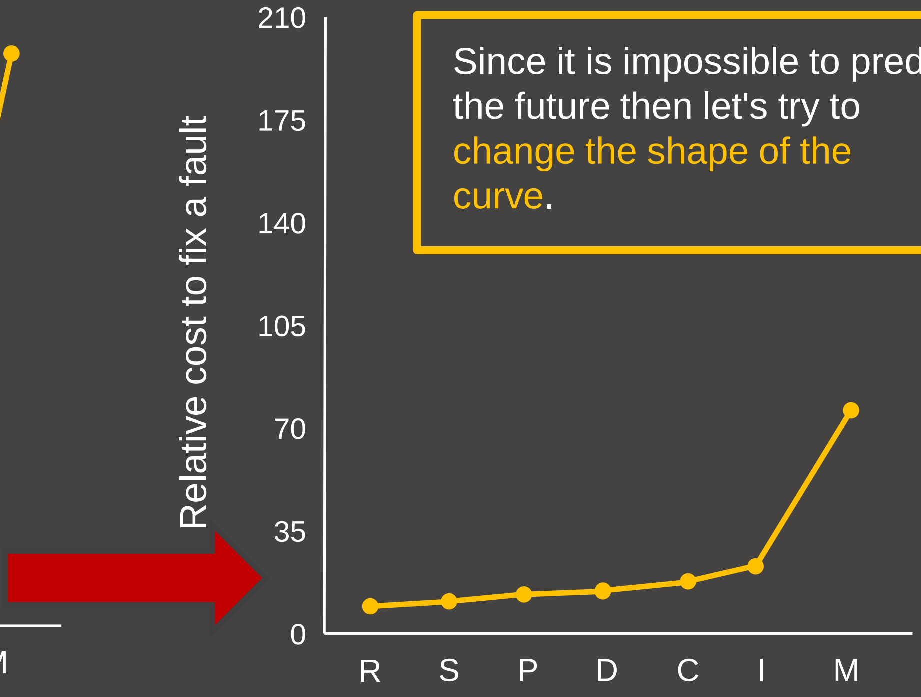
Code
Integration
Maintenance

We saw in class a graph showing the cost of fixing an error over the engineering cycle. We see that the last phases are extremely expensive and we are trying to put in place measures to prevent the workload in the last phase.

In your opinion, is it possible to avoid maintenance cycles?

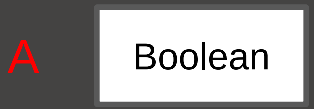
Expected engineering cycle

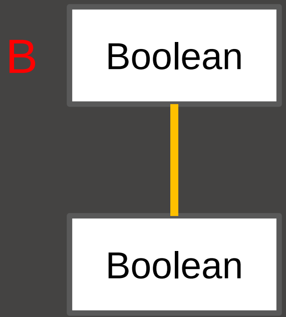


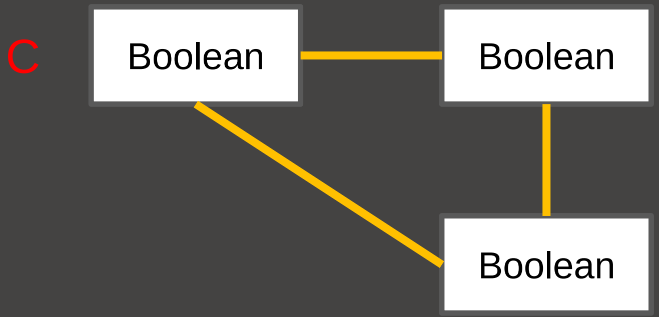


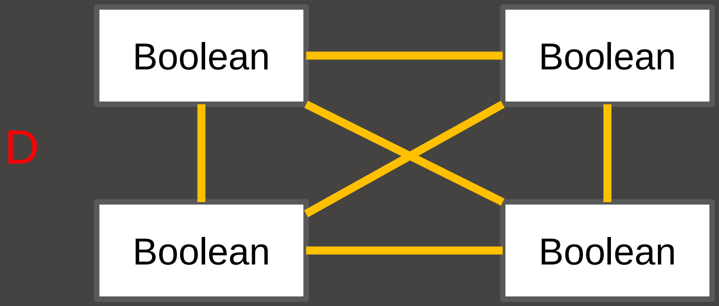
Since it is impossible to predict the future then let's try to change the shape of the curve.

The curse of dimensionality







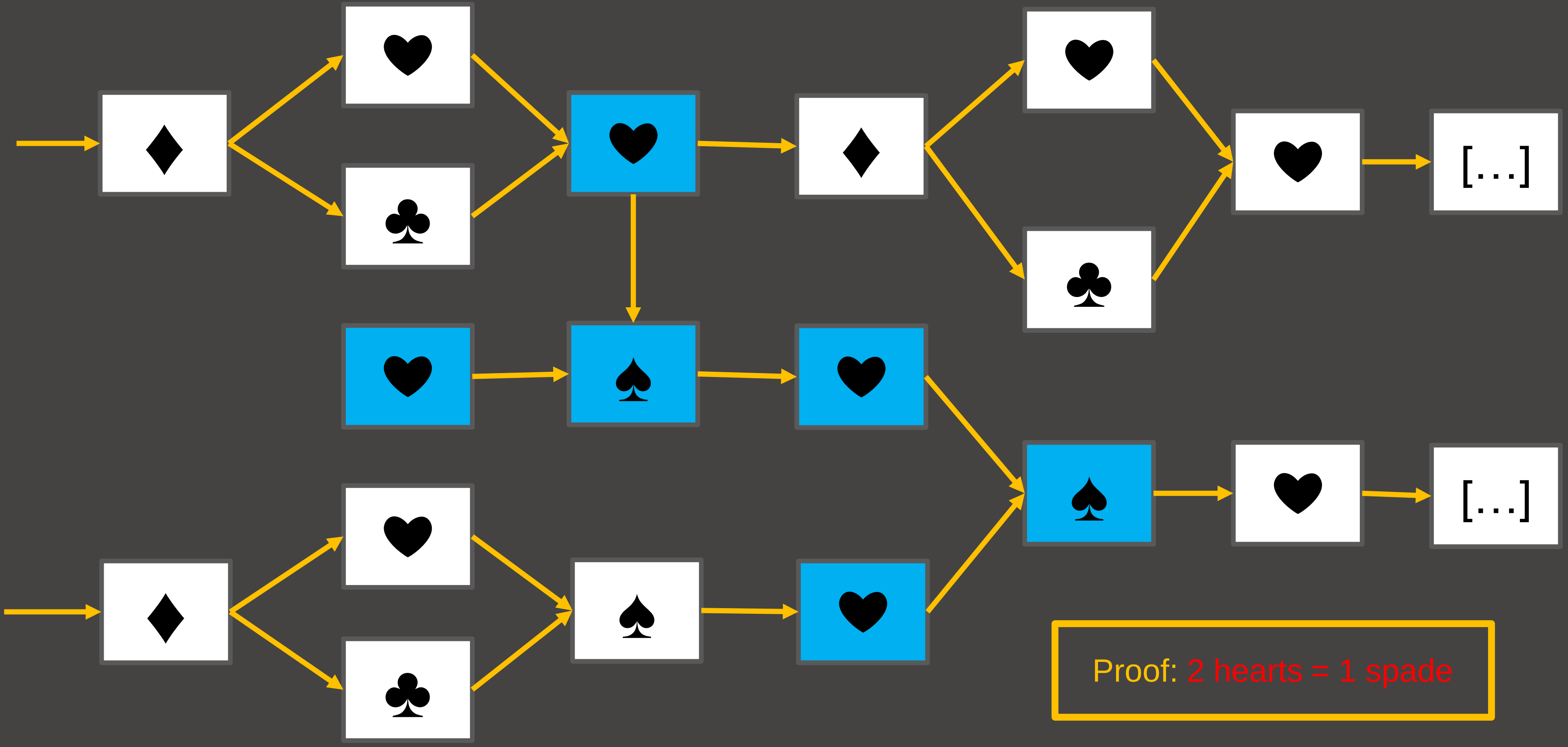


A = 2^1 = 2
B = 2^2 = 4
C = 2^3 = 8
D = 2^4 = 16

The complexity grows exponentially.

Could we fight this effect using induction?

We want symbolic computation



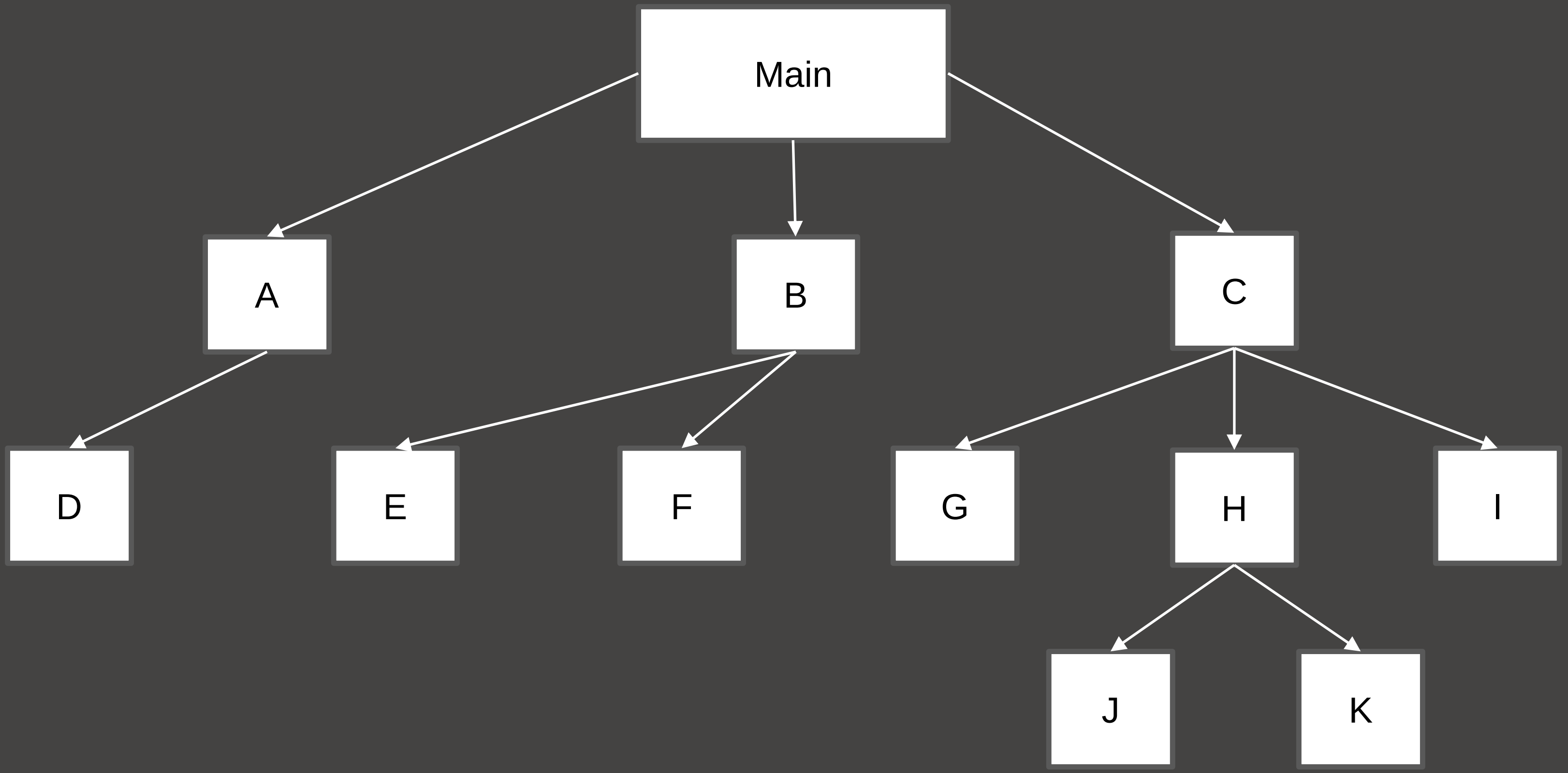
Proof: 2 hearts = 1 spade

The end of imperative

Respond quickly to market trends.

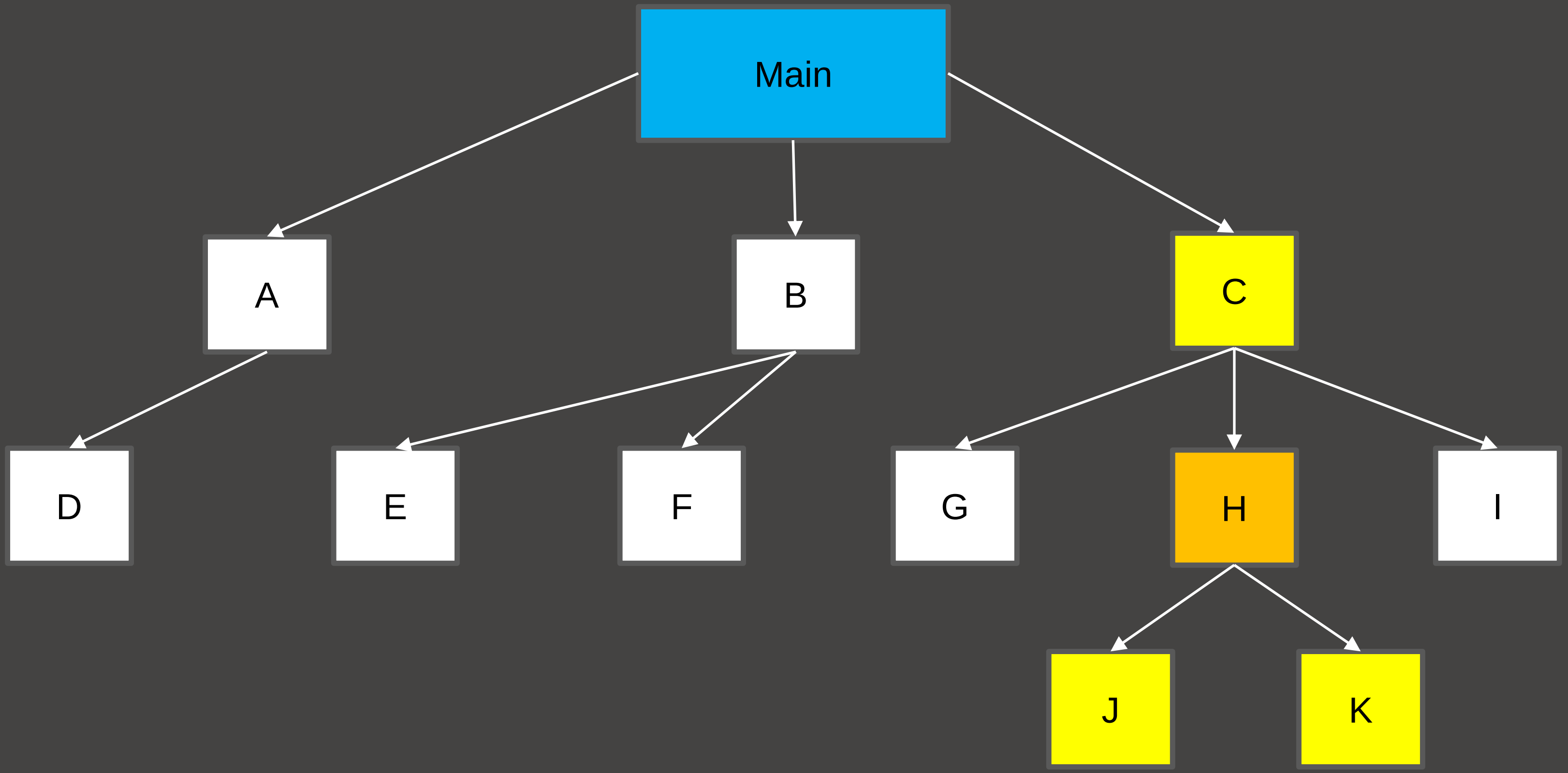
Software as a tree

H has a bad behavior, we need to implement L quickly.



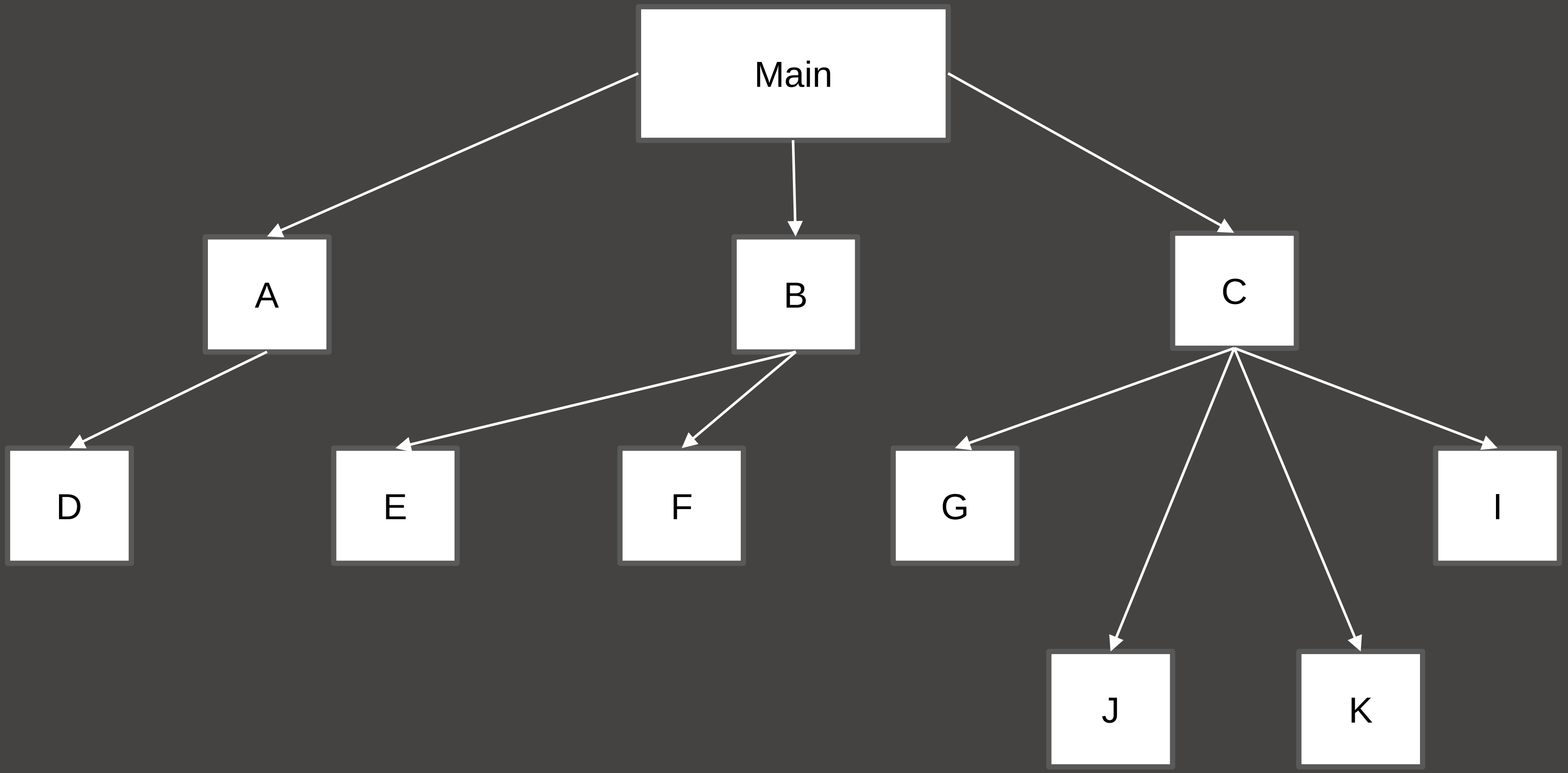
Software as a tree

Imperative iteration process



Software as a tree

The best scenario would be to delete H. But would the program work in object-oriented paradigm?



Some intuition

Think of a structure that looks like HTML

```
<nav>
    <a href=‘#someAnchor’>
        <h1>
            <span>My title</span>
            <img src=‘path/to/some/icon.png’ />
        </h1>
    </a>
</nav>
```

```
<nav>
    <h1>
        <span>My title</span>
        <img src=‘path/to/some/icon.png’ />
    </h1>
</nav>
```

Function as an HTML structure: The idea is the same as HTML, we are looking for a structure where we can add nodes and delete without paying the consequences. Note that in this structure, we can add almost anything anywhere and we look for the same thing but this time in terms of function.

Functional approach

Currying, purity, state explosion

What is wrong with this code?

```
function deleteLastElement(X) {

     const len = X.length;

     if( len > 0 )  {
         delete X[len - 1];
     }

}
```

What is wrong with this code?

```
function deleteLastElement(X) {

    const len = X.length;

    if( len > 0 )  {
        delete X[len - 1];
    }

}
```

It mutates the state!!

What is state mutation?

```
function main() {

    X = [‘a’, ‘b’, ‘c’]
    console.log(X);
    // [‘a’, ‘b’, ‘c’]

    deleteLastElement(X);
    console.log(X);
    // [‘a’, ‘b’]

    deleteLastElement(X);
    console.log(X);
    // [‘a’]
}
```

You cannot use inductive evidence given by symbolic computation because your function does not generate predictable outputs.

Even worse

```
function main() {
    [… some code …]

    const numberOfSubscribers =
          subscribers.length;
    register(subscribers);

      [… some code involving
         numberOfSubscribers …]
}
```

```
function register(subscribers) {

    [… some code …]

    deleteLastElement(subscribers);

    [… some code …]
}
```

Is it still intuitive that the list that I pass in parameter will be modified?

Notion of purity

01 **Predictive result**

Same input, same output

02 **Without interference**

Depend only on arguments passed in

03 **Without side effects**

The effects live only in the function scope

How could we fix that?

```
function deleteLastElement(X) {

     const len = X.length;

     if( len > 0 )  {
         delete X[len - 1];
     }

}
```

Other benefits gained through filtering?

```
function deleteLastElement(X) {

    const len = X.length;

    if( len > 0 )  {
        delete X[len - 1];
    }

}
```

```
function deleteLastElement(X) {

    const index = X.length - 1;

    return X.filter(
        (el, idx) => idx < index
    );

}
```

Other benefits gained through filtering?

```
function deleteLastElement(X) {

    const len = X.length;

    if( len > 0 )  {
        delete X[len - 1];
    }

}
```

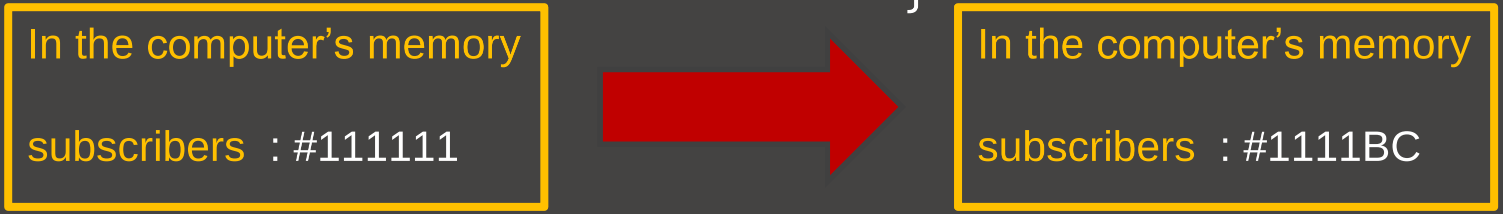
```
function deleteLastElement(X) {

    const index = X.length - 1;

    return X.filter(
        (el, idx) => idx < index
    );

}
```

Retro Engineering: Think of you doing a search on a mobile app, you enter the keywords and you are not satisfied with the results, so you enter others words. If you have filtered, you do not even need to restart the process of getting the data, you are already ready to answer the query.

What happens if the mutation is desired?

```
function register(subscribers) {

    [… some code …]

    Y = deleteLastElement(subscribers);

    [… some code …]
    return Y;
}
```

```
function main() {
    [… some code …]

    const numberOfSubscribers =
          subscribers.length;
  subscribers =
       register(subscribers);

     [… some code involving
        numberOfSubscribers …]
}
```



In the computer’s memory

subscribers : #111111

In the computer’s memory

subscribers : #1111BC

Why is this a problem?

```
function add(arr) {

  let sum = 0;

  for (const i = 0;
       i < arr.length; i++ )  {
      sum += arr[i];
  }

  return sum;

}
```

How could we fix that?

```
function add(arr) {

  let sum = 0;

  for (const i = 0;
       i < arr.length; i++ )  {
      sum += arr[i];
  }

  return sum;

}
```

Over-specifications!!

Does it change anything if I iterate in another way?

If something is not important, do not specify it. You do not improve the program, you constrain it and make it less flexible.

In agreement with Robert C. Martin, a programmer spends 80% of his time reading code. Let's optimize this time by going to the basics. There the loop was not big but in a real case, it would surely be.

How could we fix that?

```
function add(arr) {

  let sum = 0;

  for (const i = 0;
       i < arr.length; i++ )  {
      sum += arr[i];
  }

  return sum;

}
```

```
function add(arr) {

  return arr.reduce(
    (sum, num) => sum + num,
  0);

}
```

What is wrong with this code?

```
function merge(artists, artist)
{

  const name = artist.name;
  artists[name] = artist;

}
```

How could we fix that?

```
function merge(artists, artist)
{

  const name = artist.name;
  artists[name] = artist;

}
```

It mutates the state!!

Tip: If a function has no return, it has a strong chance of being imperative.

How could we fix that?

```
function merge(artists, artist)
{

  const name = artist.name;
  artists[name] = artist;

}
```

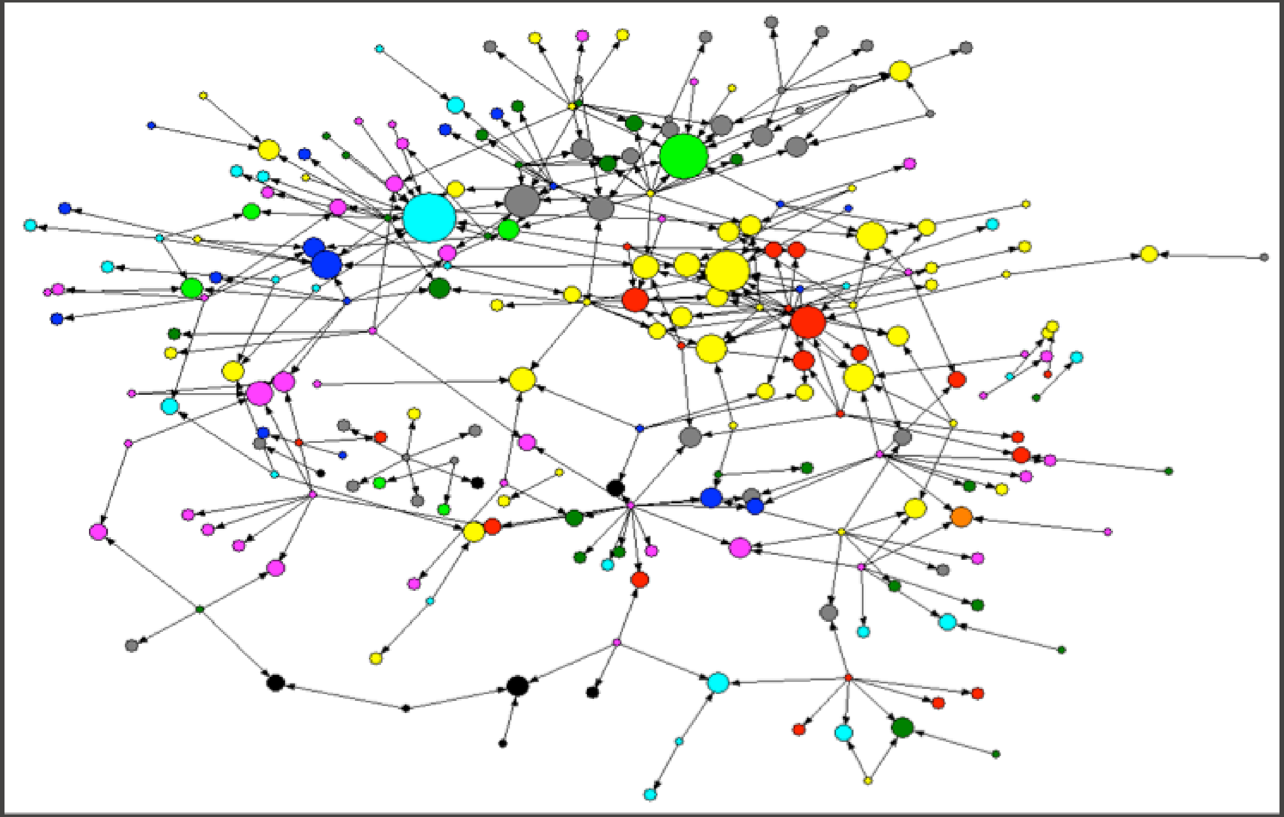
```
function merge(artists, artist)
{

  return {
    …artists,
    [artist.name]: artist
  };

}
```

The lambda architecture

Breaking the dependencies,
splitting the truth

What Object-Oriented code looks like

(most of the time)



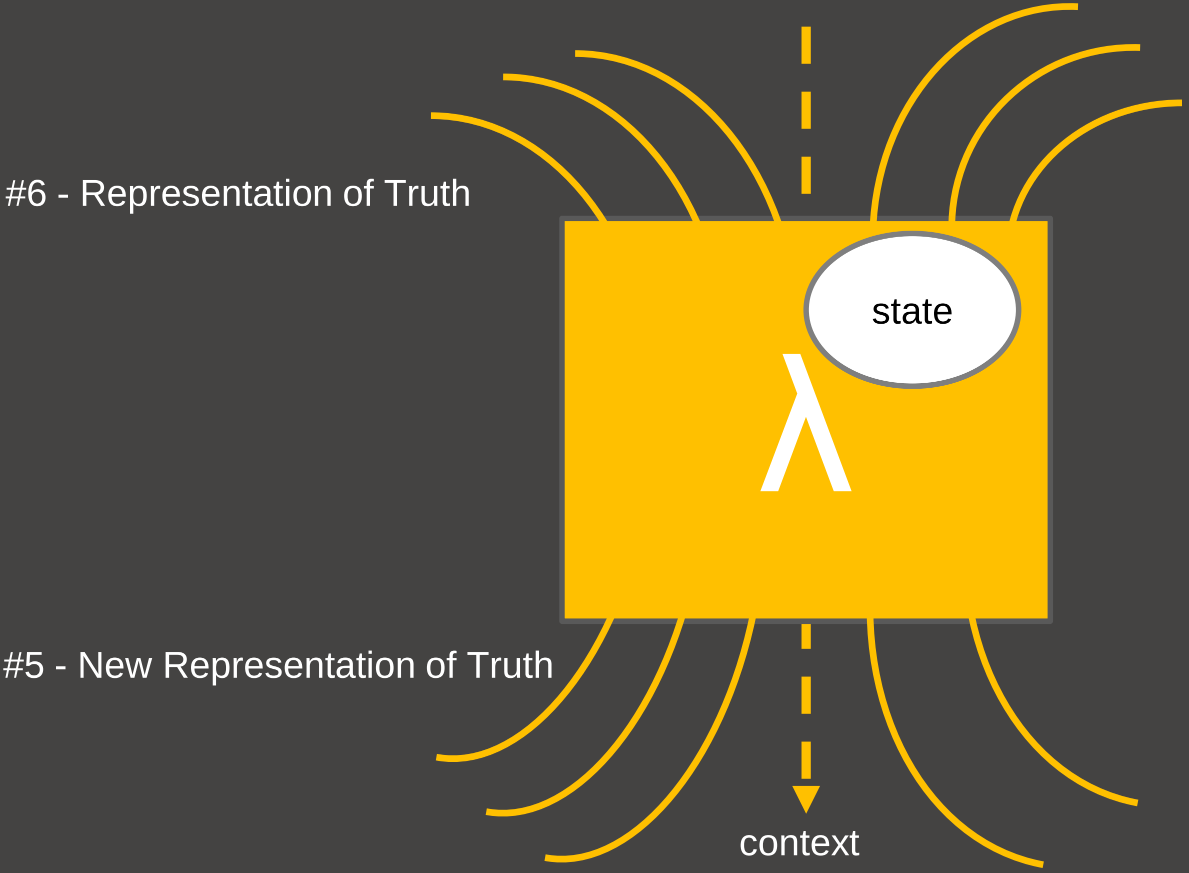
Try to reuse the code of one class for another program. Is it possible to do it without copying and pasting the content?

The problem is that every time we want to reuse we can only abstract or duplicate.

Abstracting reduces the simplicity of the code.

Duplicating reduces the efficiency of maintenance.

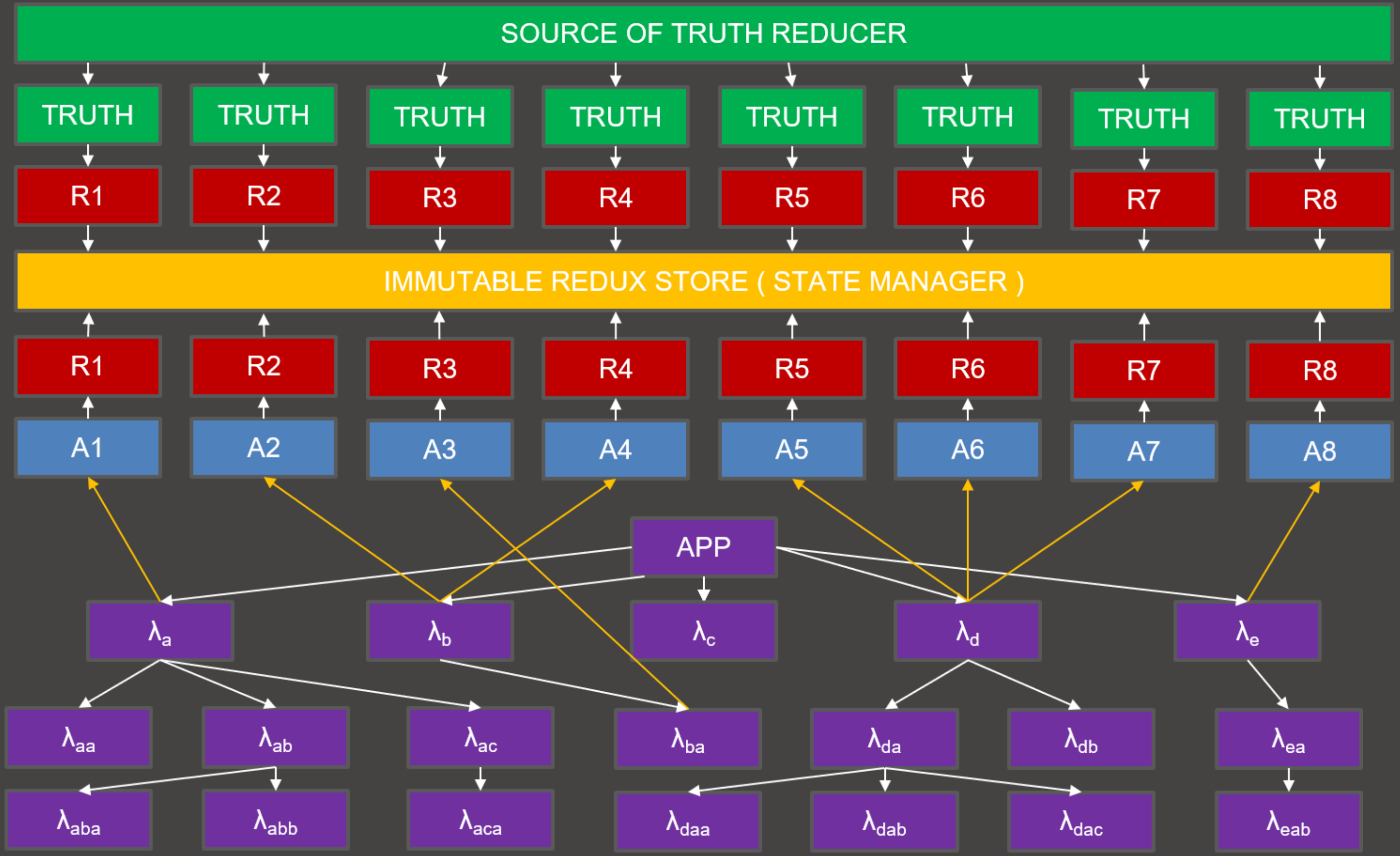
Breaking the dependencies


#6 - Representation of Truth
state
λ
#5 - New Representation of Truth
context

What we really want to do is to create an architecture that takes a series of truth parameters and returns a more granular truth by interpreting only what is in its field of knowledge.

You can see the problem as a group of people. If I ask a question, I do not expect everyone to know the answer, but I expect someone to react and others to be more expressive.

Lambda-Machine

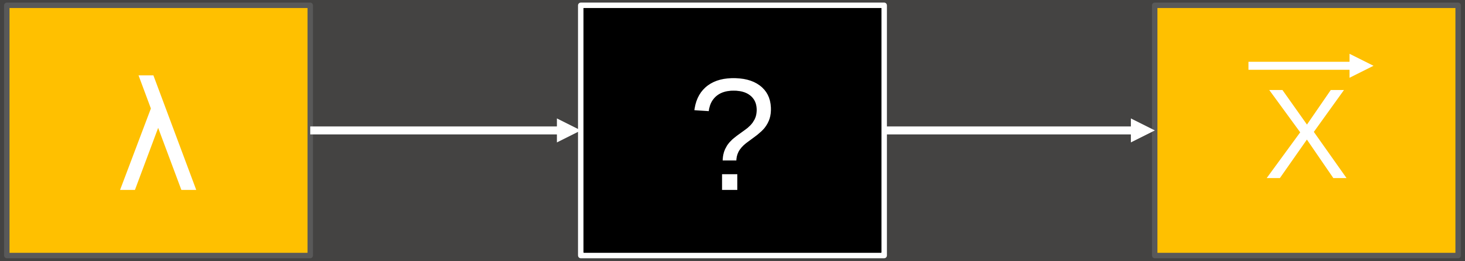


Here is a classic lambda architecture. We take a proposition and refine it until it becomes atomic proposition.

Then the atomic propositions are distributed to the lambda-nodes who will take care of interpreting it by redefining the domain of knowledge.

More in my paper.

Group Reaction Effect



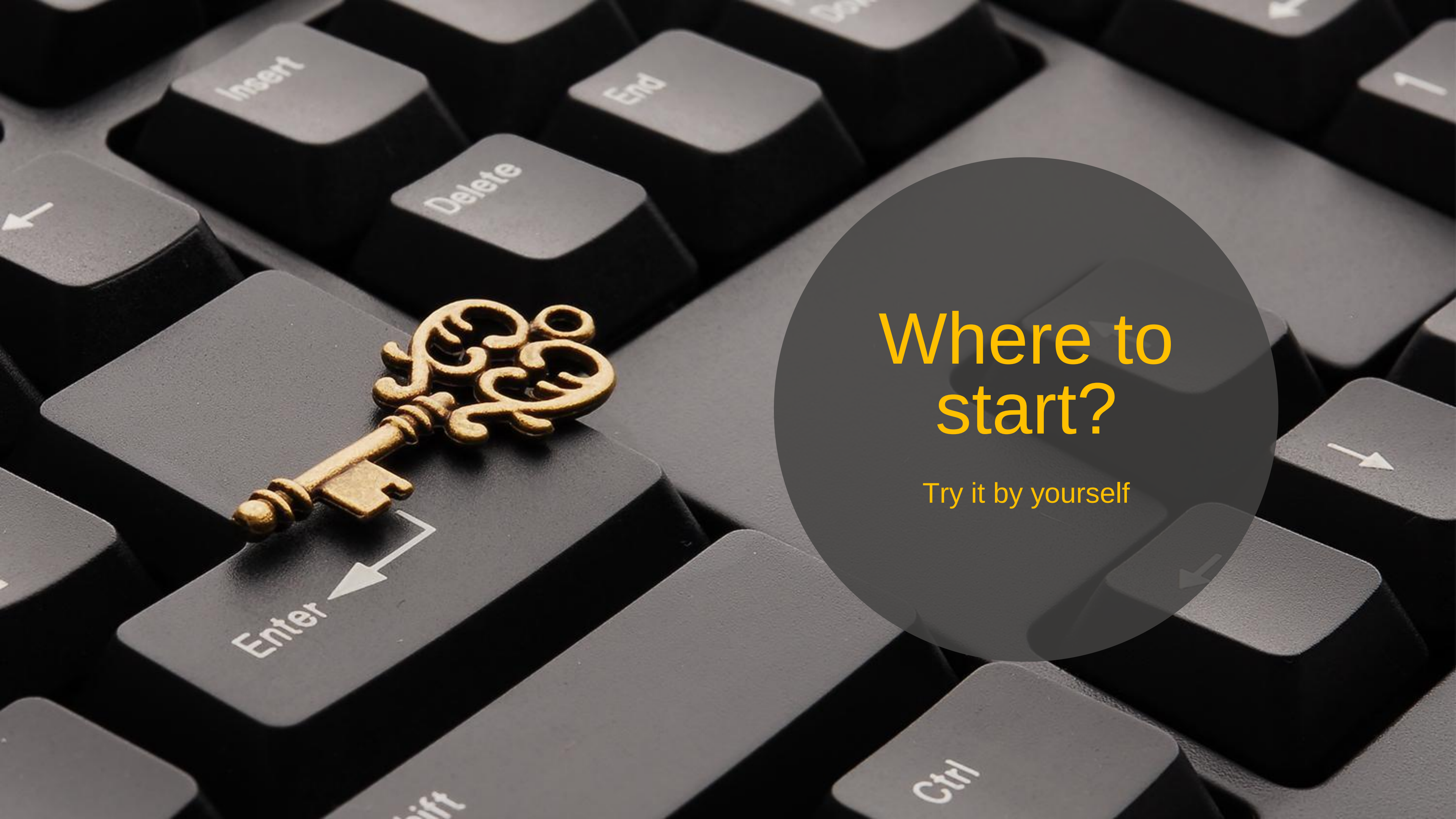
Unpredictable reaction in a black box

Splitting the truth:

The most difficult part is to convey one's intention and blindly believe in the ability of our group.

Guarantee Output:

You do not know how the elements will react with each other, but you know that they will work towards a reasonable solution.



Where to start?

Try it by yourself

Tutorials / Documentations

All the ideas expressed in this presentation are imbued with my personal and professional experiences during my computer career. If you are interested in the subject, consult the following sites to get started.

Functional Declarative

React Documentation
https://reactjs.org/docs/hello-world.html

React good tutorial
https://reactjs.org/tutorial/tutorial.html
https://reactjs.org/docs/thinking-in-react.html

Lambda Architecture

Redux
https://redux.js.org/

Redux good tutorial
https://redux.js.org/docs/basics/ExampleTodoList.html
https://spapas.github.io/2016/03/02/react-redux-tutorial/

The links offered are oriented web development because these languages have evolved and support these paradigms very well. Several other languages also cover the same subject, but they are rarely derived to make them complete programs. **LINQ**, **F#**, **SQL**, **Lodash**, **Java 8**, etc.

Credits

This presentation template created by GoogleSlidesppt.com

Some shapes and icons in this template were created by GoogleSlidesppt.com

Some backgrounds in this template were created by GoogleSlidesppt.com

Some illustrated images in this template were created by GoogleSlidesppt.com

Some inserted pictures in this template were created by pixabay.com

Thank you for using GoogleSlidesppt.com