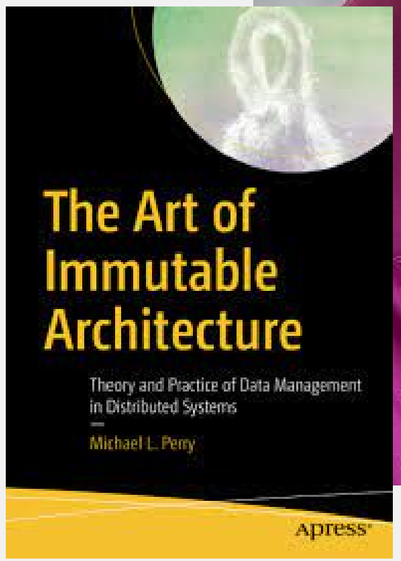
IMMUTABLE ARCHITECTURE

David Martínez Castañón
Iván Menéndez Mosegui
Jorge Joaquín Gancedo Fernández

Index

Michael L. Perry

- Software architect and consultant
- Specialized in large-scale distributed systems
- Author of several books on software architecture and design
- Founder and principal consultant at Clear Measure
- Active member of the software development community



Immutable architecture

Software design approach that focuses on the use of immutable data structures. Instead of modifying data, new copies are created.

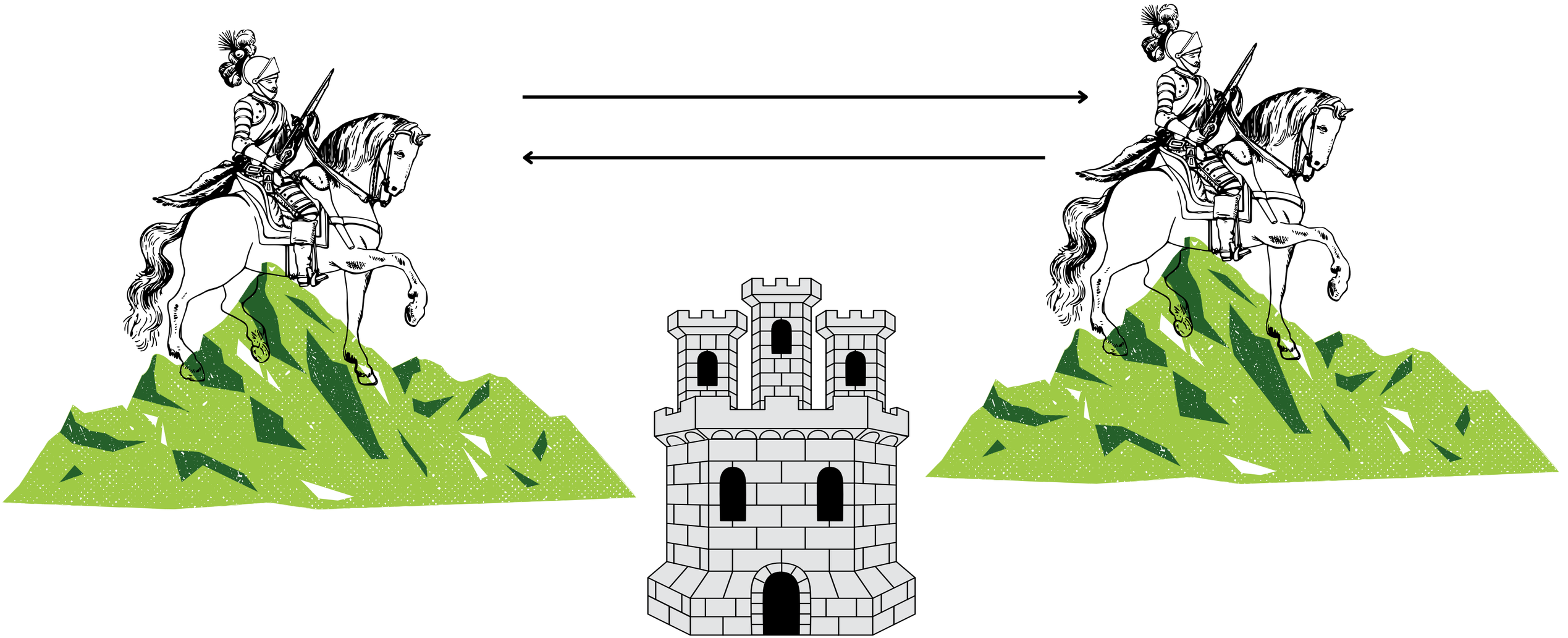
Advantages	Disadvantages
• Thread-safe • Easier to understand and test • Reliability • Reuse of data structures	• Could be less efficient • More difficult to implement • High memory consumption • Not always feasible

Immutable Architecture against Immutable Infrastructure

Immutable Architecture	Immutable Infrastructure
• Architectural paradigm (how the software is built). • *Immutable* does not mean it can't change.	• Deployment paradigm (how the software is deployed). • Solutions never change in their lifetime, but new versions may be built on top of them. • Solutions are replaced when their configuration must change.

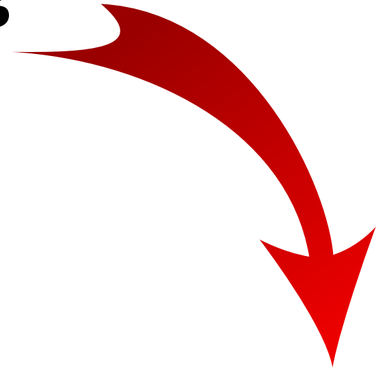
Immutable architecture and the Two Generals' Problem

The Two Generals' Problem



The Two Generals' Problem

The two generals' problem is in fact an unsolvable networking problem.



But its consequences are universally acknowledged in computing! Remember the CAP theorem?

The Two Generals' Problem and Immutable Architecture

Michael L. Perry first thought about immutable architecture when he was writing a distributed gift card system. He did not know how to achieve consistency between the different systems.

The systems were the generals, and achieving consistency was attacking the castle.

The Two Generals' Problem and Immutable Architecture

Immutable architecture forgoes consistency, prefering to achieve *strong <u>eventual</u> consistency*

"The generals will attack at some undefined point in the future"

Historical Modeling

Record changes to a system, instead of its state

Definition

"Historical Modeling is a method of distributed smart client software construction. It is based on a model of software behavior as a graph of partially ordered facts."

Michael L. Perry

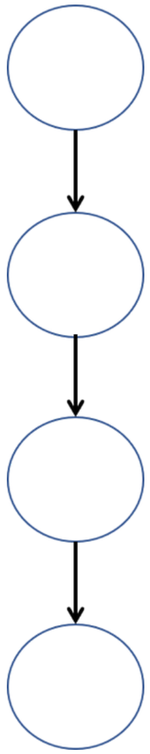
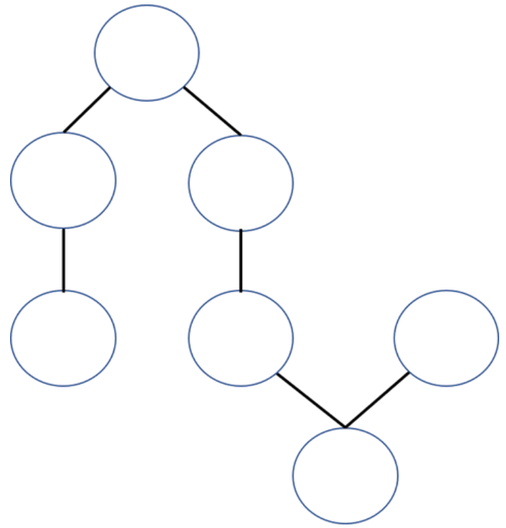
Event sourcing vs historical facts

Event sourcing

- Fully ordered (chronologically)
- Completely independent

Facts

- Partially ordered (parents)
- Link to their predecessor(s)

Event sourcing	Historical facts
$s_k = E(s_{k-1}, e_{k-1}) / k = \{1, \ldots, n\}$ 	

Steps of the process

1. Identify the changes

Register user actions that modify the system.

2. Refine changes

Put together changes, pull out entities and register constraints.

3. Query the model

Develop queries that answer the user's questions when making changes.

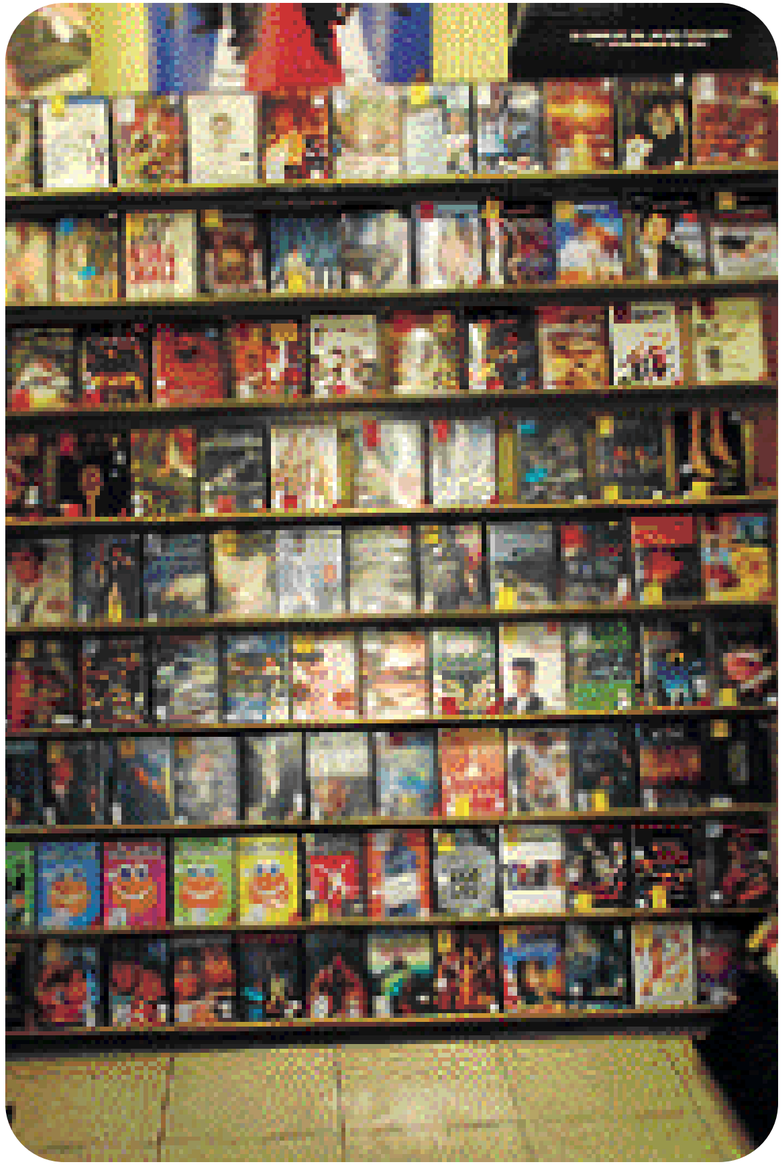
4. Repeat

Answer the remaining questions.

Videoclub

The customer can...

- Rent a film
- Buy a film
- Return a film
- List the films of a certain genre



1. Identify the changes

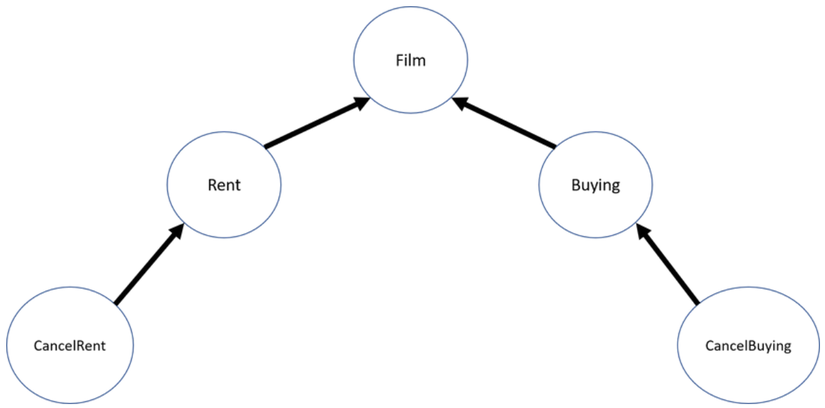
Listing

- Rent a film
- Buy a film
- Return a film
- List the films of a certain genre

Rewriting

- Rental (Film film, date rentalDate, double rentalAmount)
- Buying (Film film, date rentalDate, double rentalAmount)
- Return (Buying buying)
- CancelRent (Rent rent)
- CancelBuying(Buying buying)

Representing



2. Refine changes

Group entities

- Adquisition (Film film, date adquisitionDate, double amount)
  - Rental()
  - Buying()
- CancelAdquisition (Adquisition adquisition)

Pull out entities

- Film (string name)

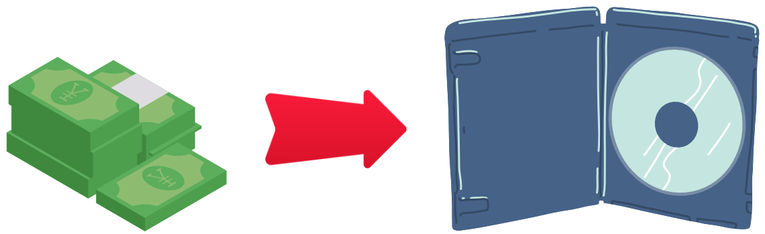
Prerequisites

- *"It should be possible to apply the VAT to each adquisition"*
- VAT(Adquisition adquisition).

3. Query the model

A change is performed

- A film is bought



We will record a new...

- Adquisition(Film film...)

User may wonder....

Write the query

```
fact Adquisition {
  Film film;
  Date adquisitionDate;
  double amount;

  bool isCancelled {
    exists CancelAdquisition cancel :
        cancel.cancelledAdquisition = this
  }
}
```

4. Repeat

There may be more prerequisites and changes to take into account...

- **More users**: What about the workers? Is there a manager?
- **Business rules**: A customer may not rent more than 3 films simultaneously

What do we do?

What we have done until now:

- Identify new changes
- Refine them
- Write the appropiate queries

It is an iterative process!

Thank you!

Do you have any questions?