COMMUNITYDAY

CLOJURE – Martin Jul, @mjul

Clojure

En kraftfuldt erstatning for JavaScript, C# og Java

Community Day 2012
10. maj 2012

```
{:name "Martin Jul"
 :email "mj@ative.dk"
 :twitter "mjul"}
```

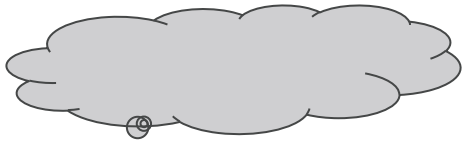
A|T|I|V|E|

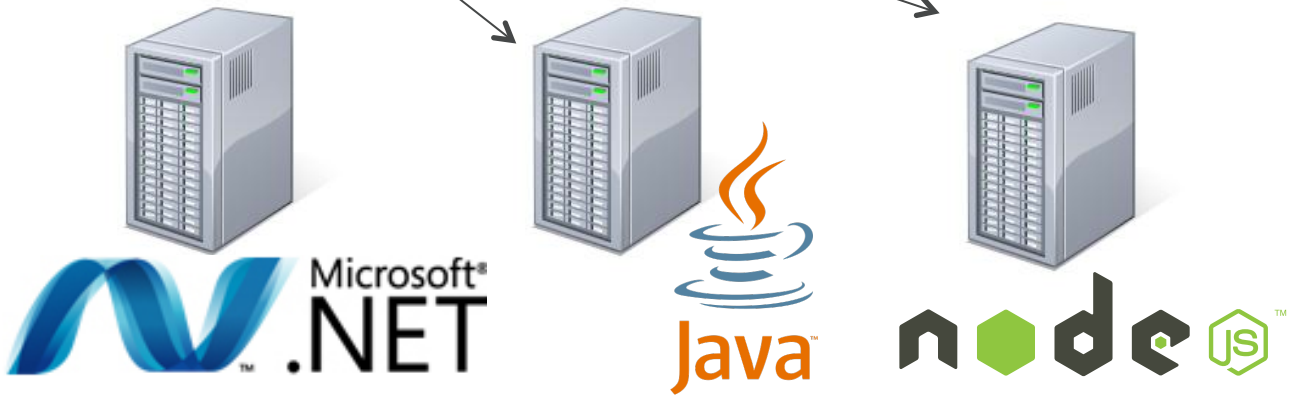
A Better Language for Everything

JavaScript targets

A modern
Lisp language

Compiler
JVM, CLR
ClojureScript
(Clojure-py...)





The main server platforms

A Better Java

```java
public class StringUtils {
  public static boolean isBlank(String str) {
    int strLen;
    if (str == null || (strLen = str.length()) == 0) {
      return true;
    }
    for (int i = 0; i < strLen; i++) {
            if ((Character.isWhitespace(str.charAt(i)) == false)) {
              return false;
            }
    }
    return true;
  }
}
```

```clojure
(defn blank? [s]
  (every? #(Character/isWhitespace %) s))
```

Example from “Programming Clojure” by Stuart Sierra (Pragmatic Programmers, 2009), p. 23

A Better JavaScript

```
var confs = [
 {name : "Clojure Conj", year : 2011 },
 {name : "Community Day", year : 2012 }];


var sorted = confs.sort(
  function(a, b)
  {
    if (a.name < b.name)
      return -1
    if (a.name > b.name)
      return 1
    return 0
  });
```

Small, Powerful and Extensible

Simple core language

Excellent **function library**

Persistent collections

Immutability by default

Sane Reasoning about State

Software Transactional Memory

Sane Concurrency

Code is data: **powerful meta programming** features

Typed Clojure Gradual typing	**core.logic** Logic programming	**Trammel** Code contracts	**Eastwood** C Lint

Simple libraries make Haskell, Prolog and Eiffel hackers feel at home

Why Clojure?

LISP is worth learning for a different reason: the profound enlightenment experience you will have when you finally get it. That experience will make you a better programmer for the rest of your days, even if you never actually use LISP itself a lot.

Eric S Raymond

"How to Become a Hacker"

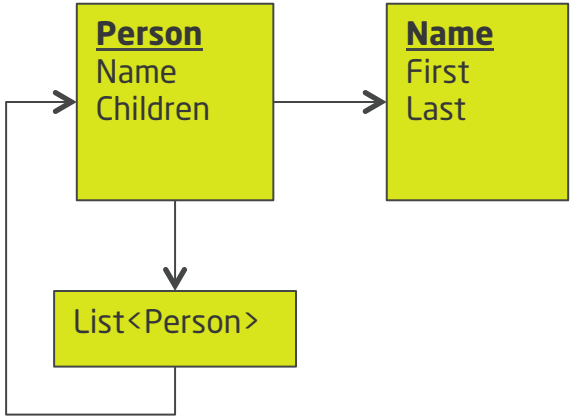
Reducing the Complexity of the Implementation Domain

Problem	Simplification
Spaghetti code	Structured programming, OO
Memory management	Garbage collection
Side-effects	**Pure functions**
Sharing data	Message passing, value semantics **Immutable data**
Concurrency / locks	**Software Transactional Memory** Message based concurrency Offline lock patterns, ...
Composability	**Common abstractions**, higher-order functions
Limitations of implementation language	**Macros** DSLs, Design patterns

MUTABLE STATE IS THE NEW SPAGHETTI CODE

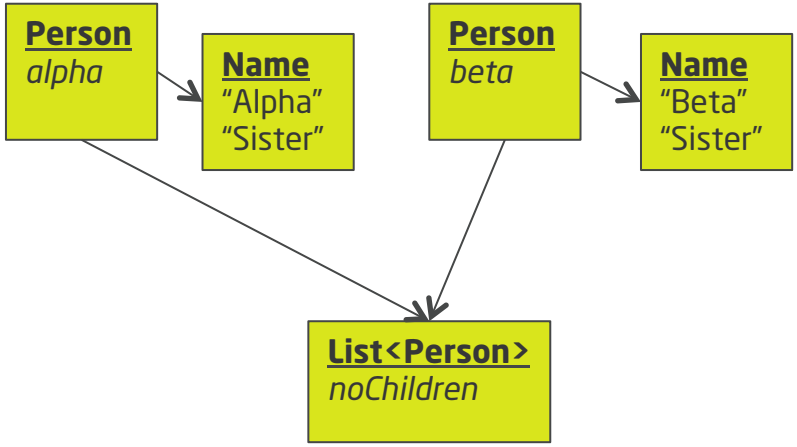
Mutable state:
What is wrong with this code?

```
// Naïve version
public class Name {
        public String First { get; set; }
        public String Last { get; set; }
}

public class Person {
        public Person(Name name, List<Person> children)
        {
            this.Name = name;
            this.Children =  children;
        }
        public Name Name { get; set; }
        public List<Person> Children { get; }
}
```



Mutable state:
What is wrong with this code?

```
var noChildren = new List<Person>();
var alpha = new Person(new Name("Alpha", "Sister"),
                       noChildren);
var beta  = new Person(new Name("Beta", "Sister"),
                       noChildren);
```

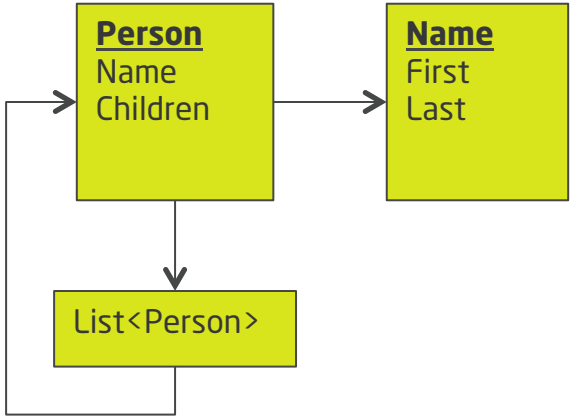



```
alpha.Name.Last = "Omega";
var gamma = new Person(new Name("Gamma", "Alphadaughter"));
alpha.Children.Add(gamma);
DoSomethingTo(alpha, beta);
```

What is the state after this?

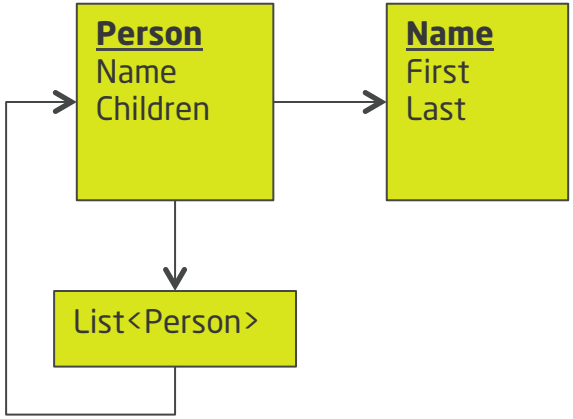
Mutable state:
What is wrong with this code?

```
// Improved

public class Name {
    public String First { get; set; }
    public String Last { get; set; }
}

public class Person {
    public Person(Name name, List<Person> children)
    {
        this.name = name.DeepClone(); // if Name is mutable
        this.children = DeepClone(children);
    }
    public Name Name { get; set; }
    public IEnumerable<Person> Children { get { return DeepClone(children); }}
    public Name UpdateName(String f, String l) { this.Name = new Name(f,l); }
    public AddChild(Person child) { this.children.Add(child.DeepClone()); }
}
```



Mutable state

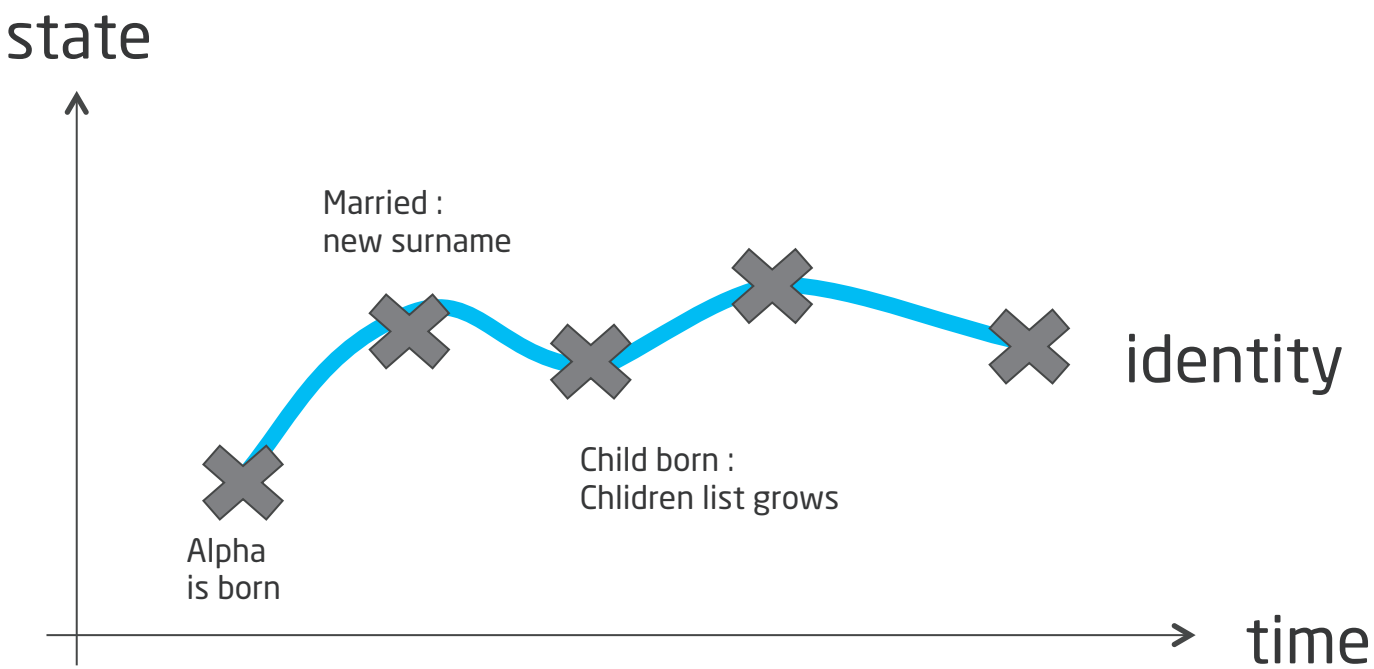
- **Encapsulation** is hard
  - clone in, clone out
- **Ownership** is hard
  - “Entities” and “Value Objects”
- **Reasoning** about state is hard
- **Concurrency** is even worse



Maybe it’s time to stop

IMMUTABILITY

Philosophy of State and Identity



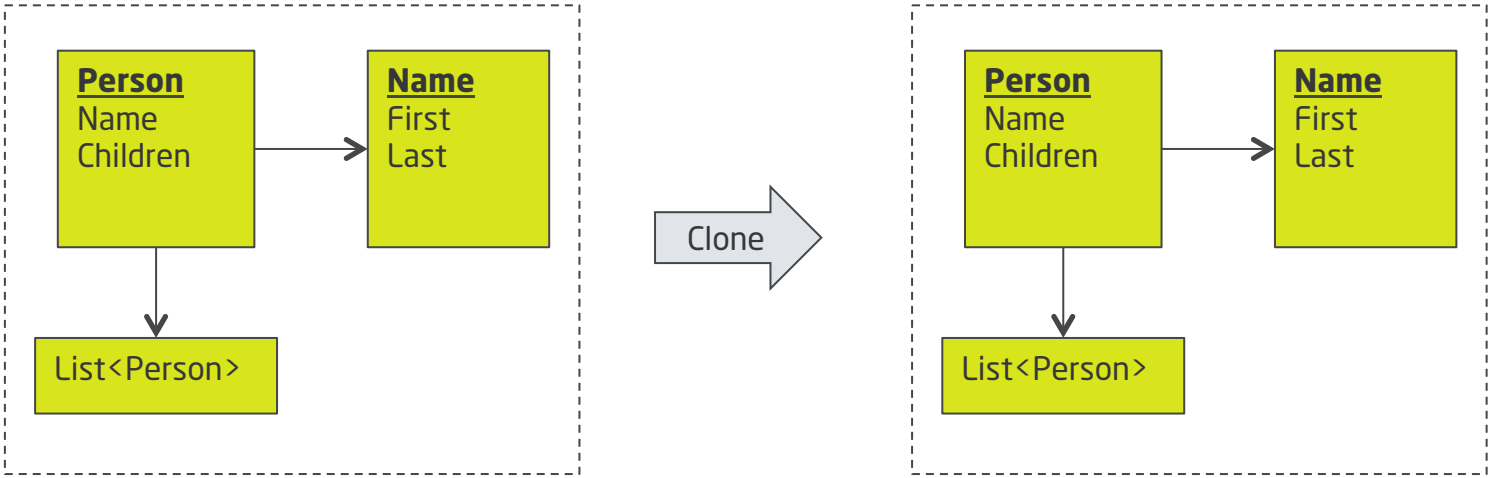
Advantages of Immutability

- Check invariants at construction only
- Reasoning about code is much simpler
- Thread safe
- Iteration safe
- No locks required

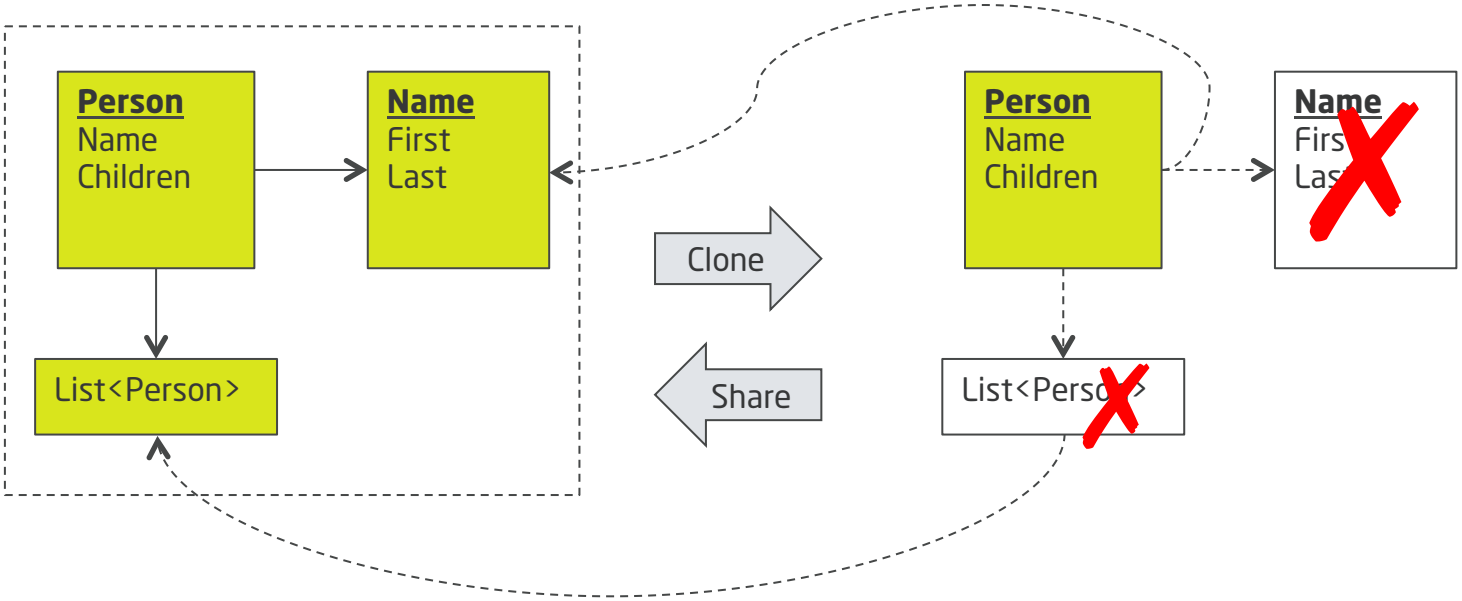
Disadvantages of Immutability

- We need a way do it efficiently
  - Memory
  - Performance
- We need a mutation mechanism

Structural Sharing

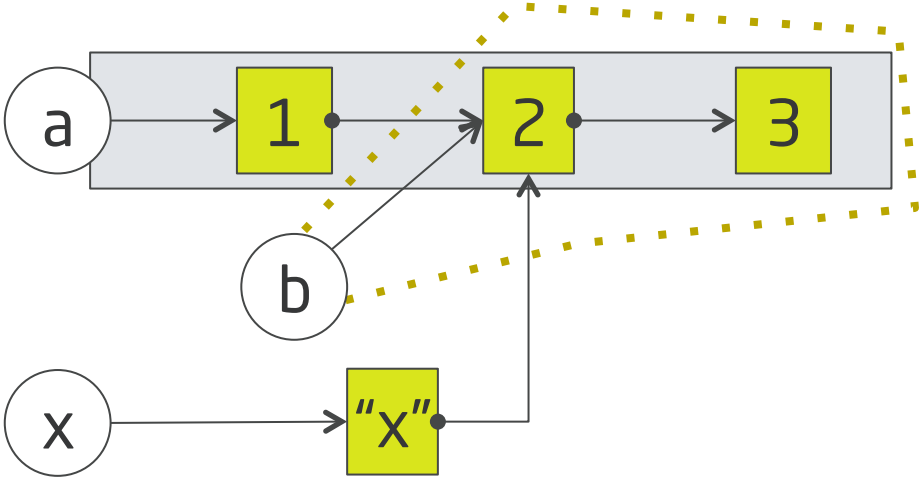


Structural Sharing



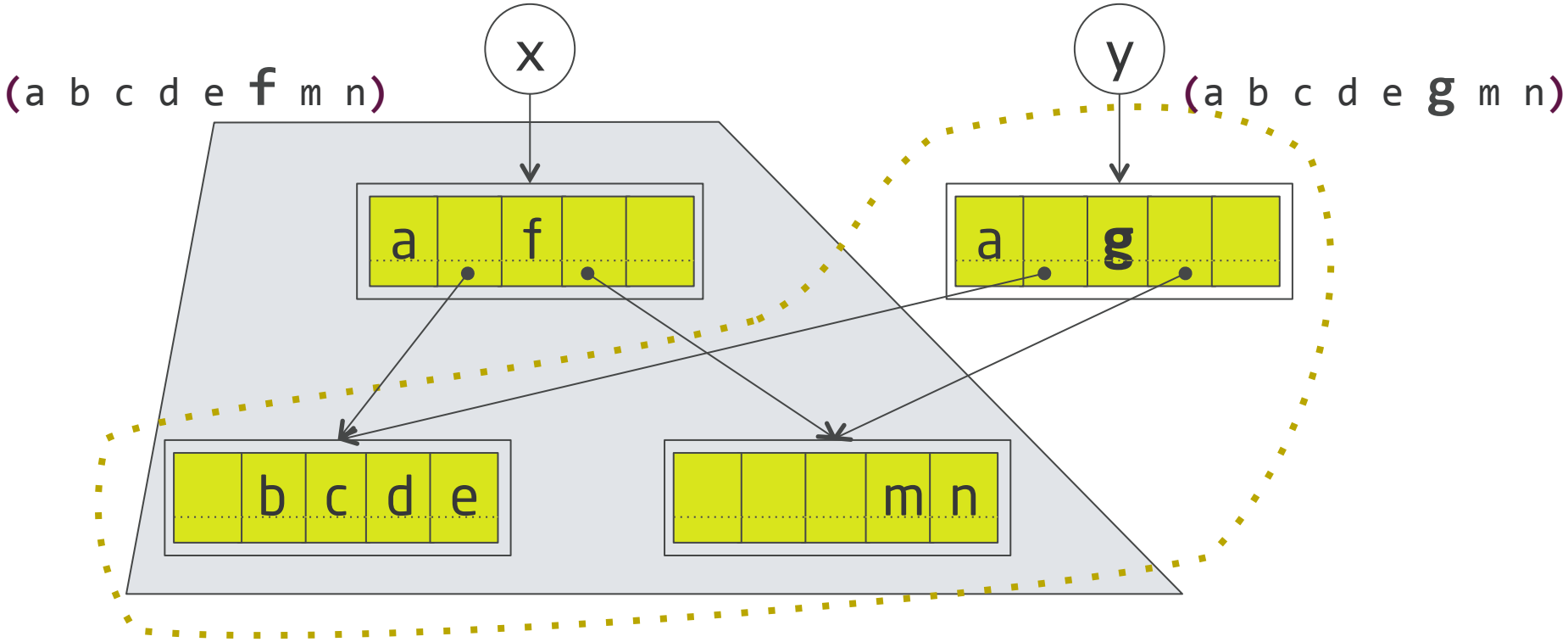
Persistent Collections for performance

```
(def a (list 1 2 3))
=> (1 2 3)

(def b (rest a))
=> (2 3)

(def x (conj b "x"))
=> ("x" 2 3)
```



- Immutable
- Structural Sharing
- Copy-on-write semantics

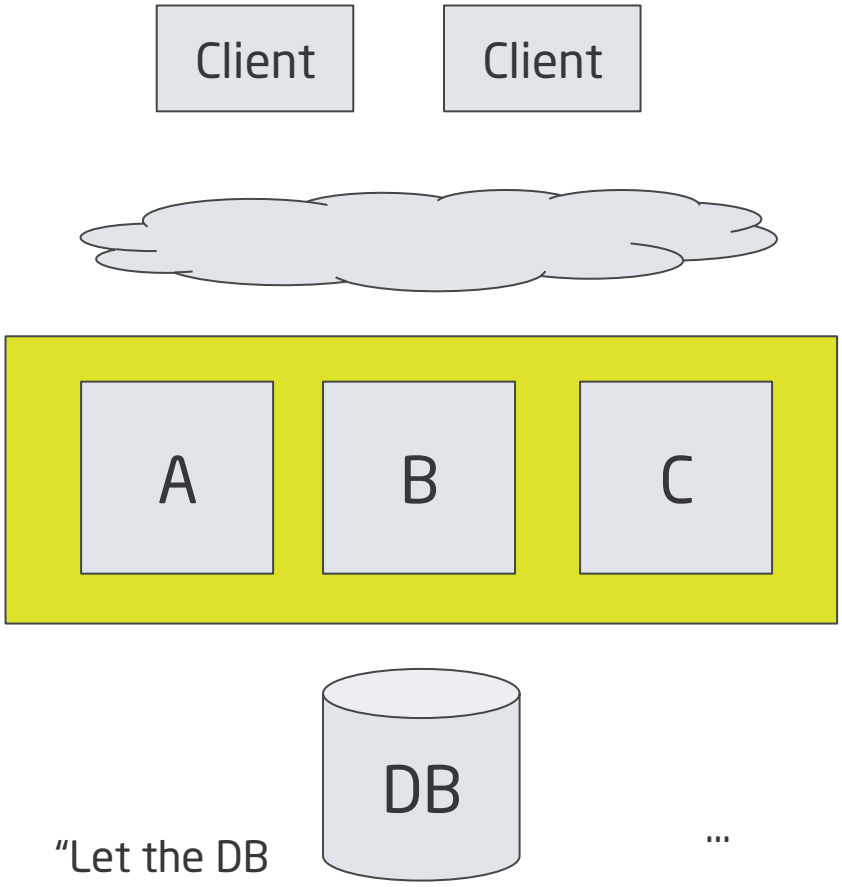
Persistent Collections implemented with hash tries



Extremely simplified diagram!
For full details see: Fast and Space Efficient Trie Searches, Bagwell [2000]

CONCURRENCY WITH SOFTWARE TRANSACTIONAL MEMORY

Concurrency Strategies



In process

Threads
Locks

Offline lock patterns

"Let the DB handle it"

Inter process

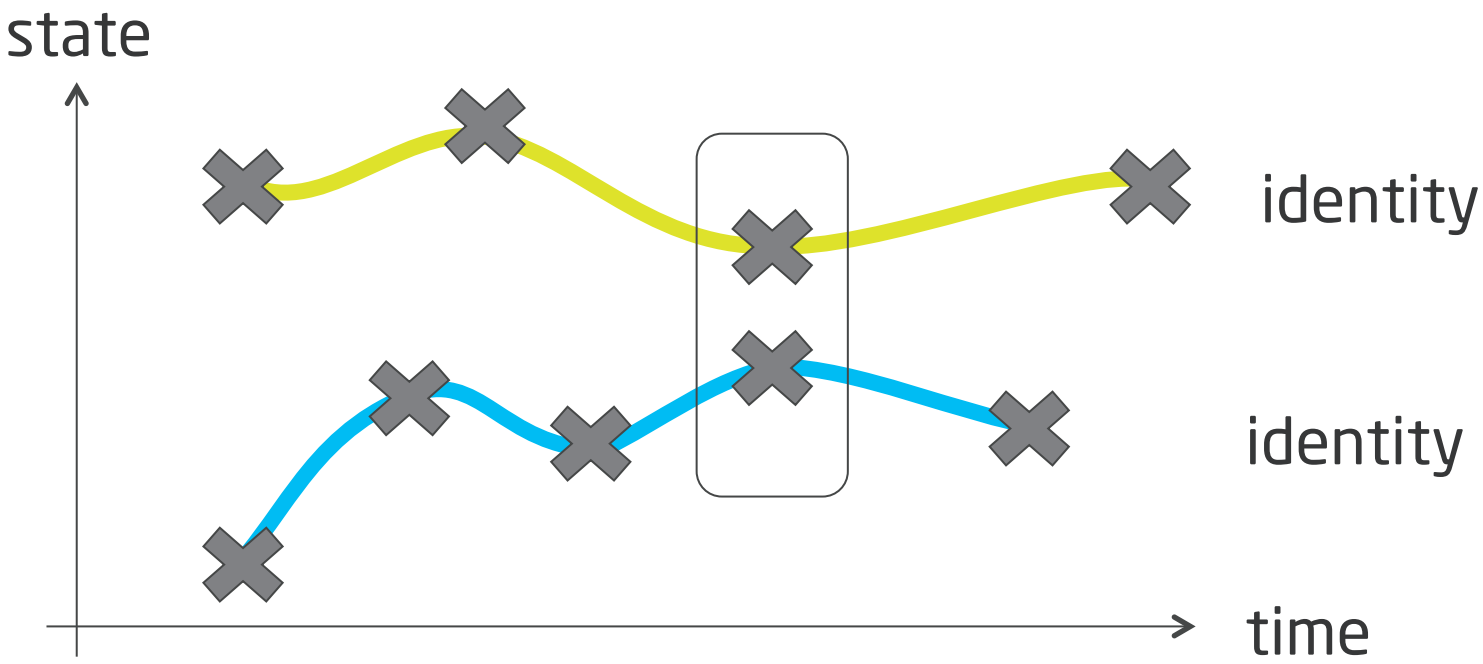
Scheduler

Actor model

Message passing

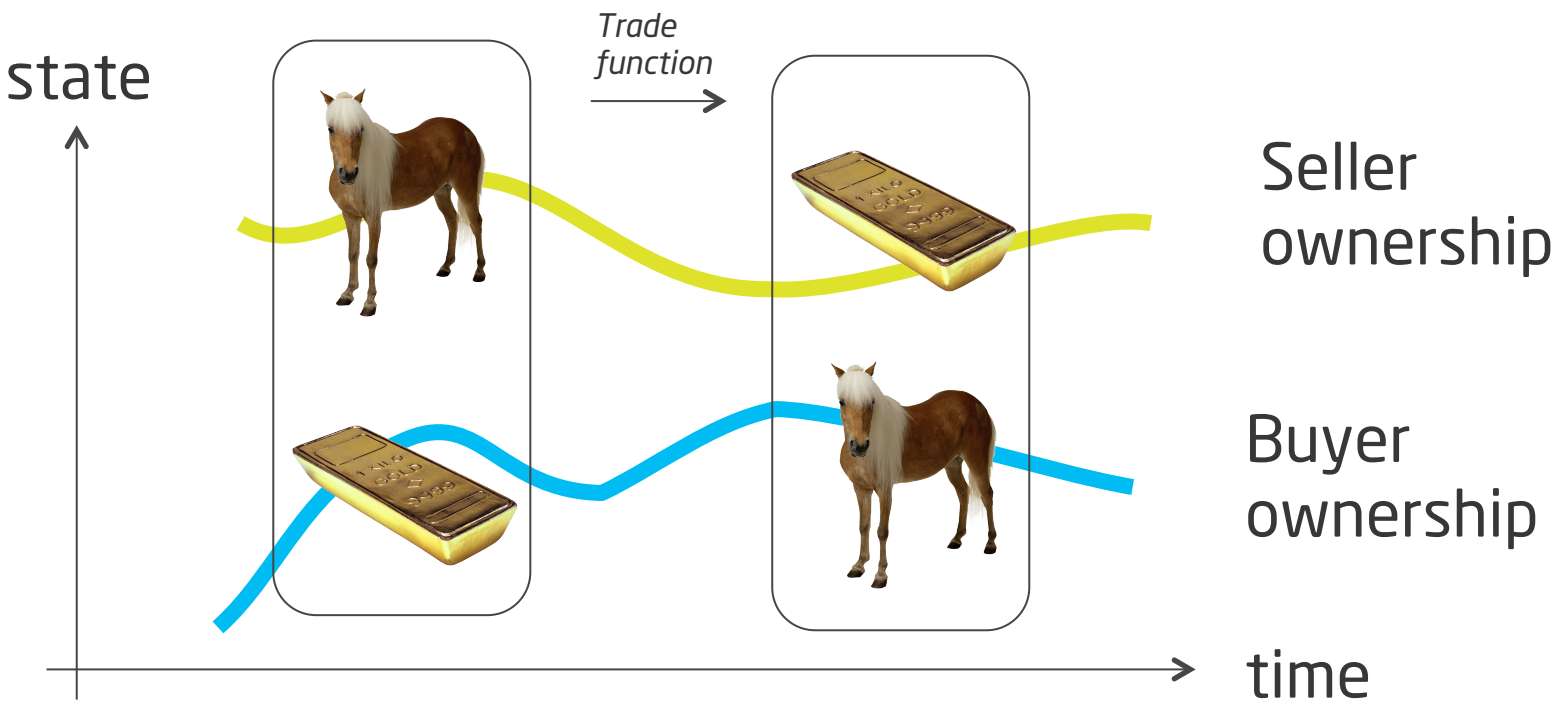
...

Clojure Concurrency

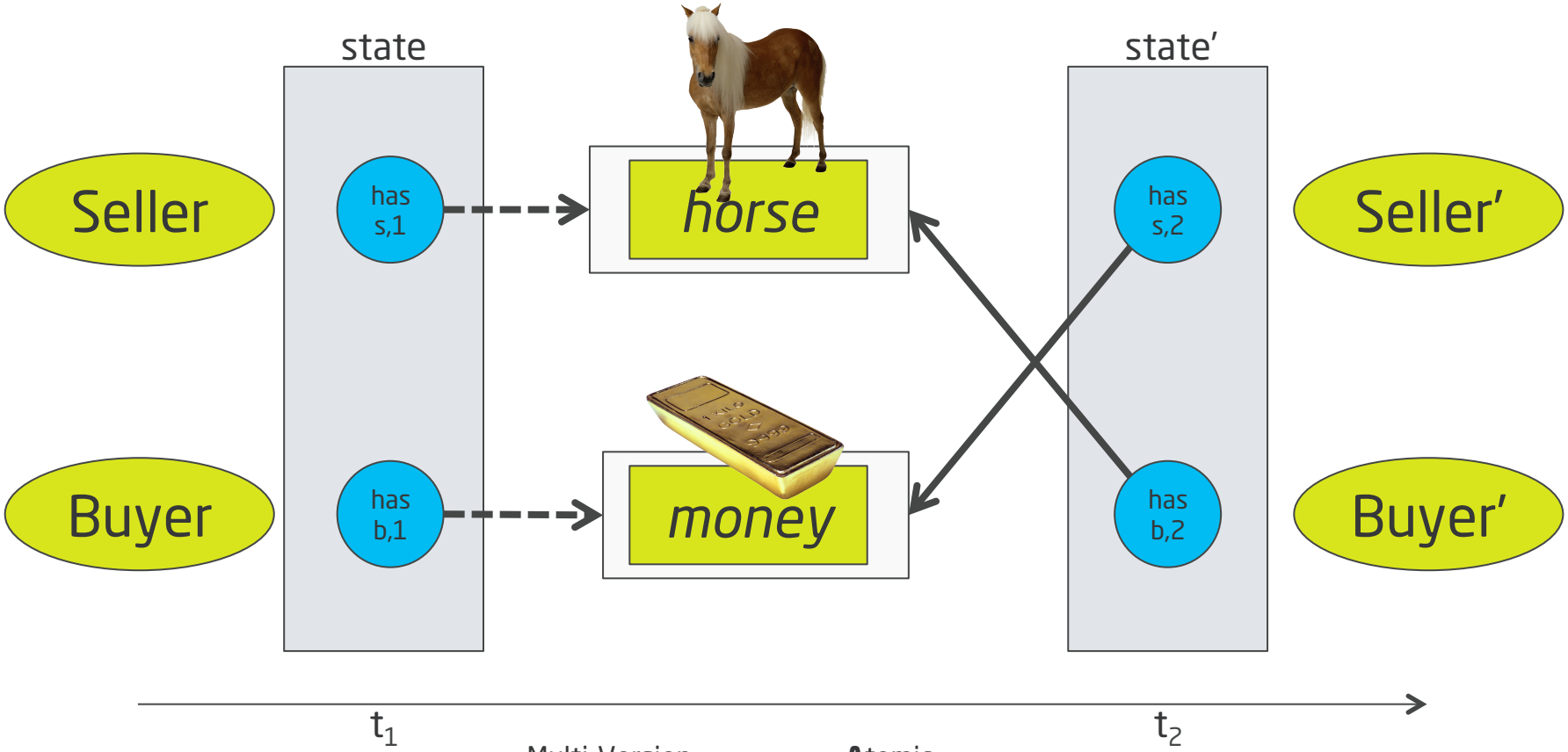


- **Indirect** references to immutable data structures
- Concurrency semantics for references
  - Automatic/enforced
  - No locks

Clojure Concurrency



Software Transactional Memory



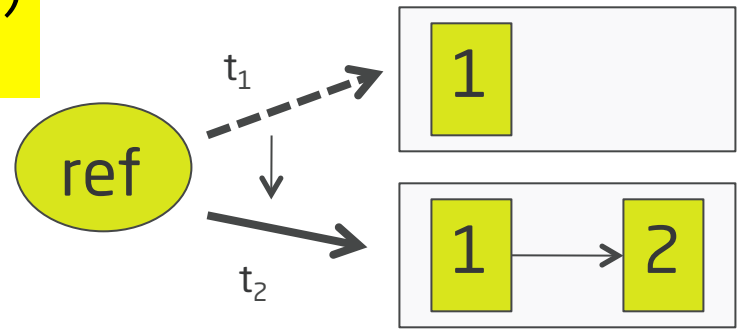
Multi-Version
Concurrency Control

Atomic
Consistent
Isolated
~~Durable~~

STM Example

```clojure
(defn post
  "Post an amount to the account."
  [account amount msg]
  (conj account {:amount amount, :msg msg}))

(defn transfer
  "Transfer an amount between two accounts."
  [from to amount msg]
  (dosync
    (alter from post (- amount) msg)
    (alter to post amount msg)))
```

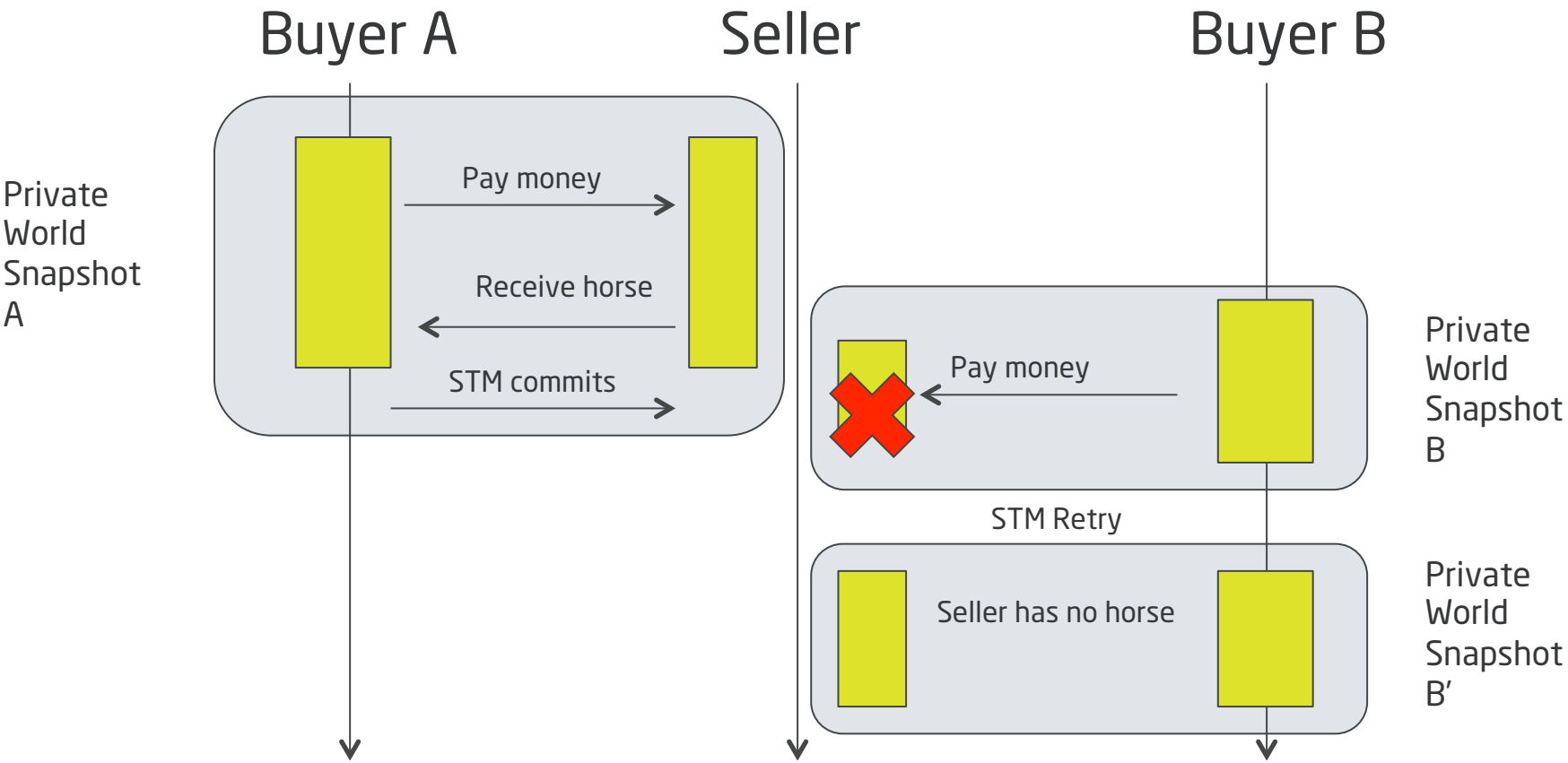
:amount	:msg
1000	Initial balance
-170	Train fare
-40	Coffee



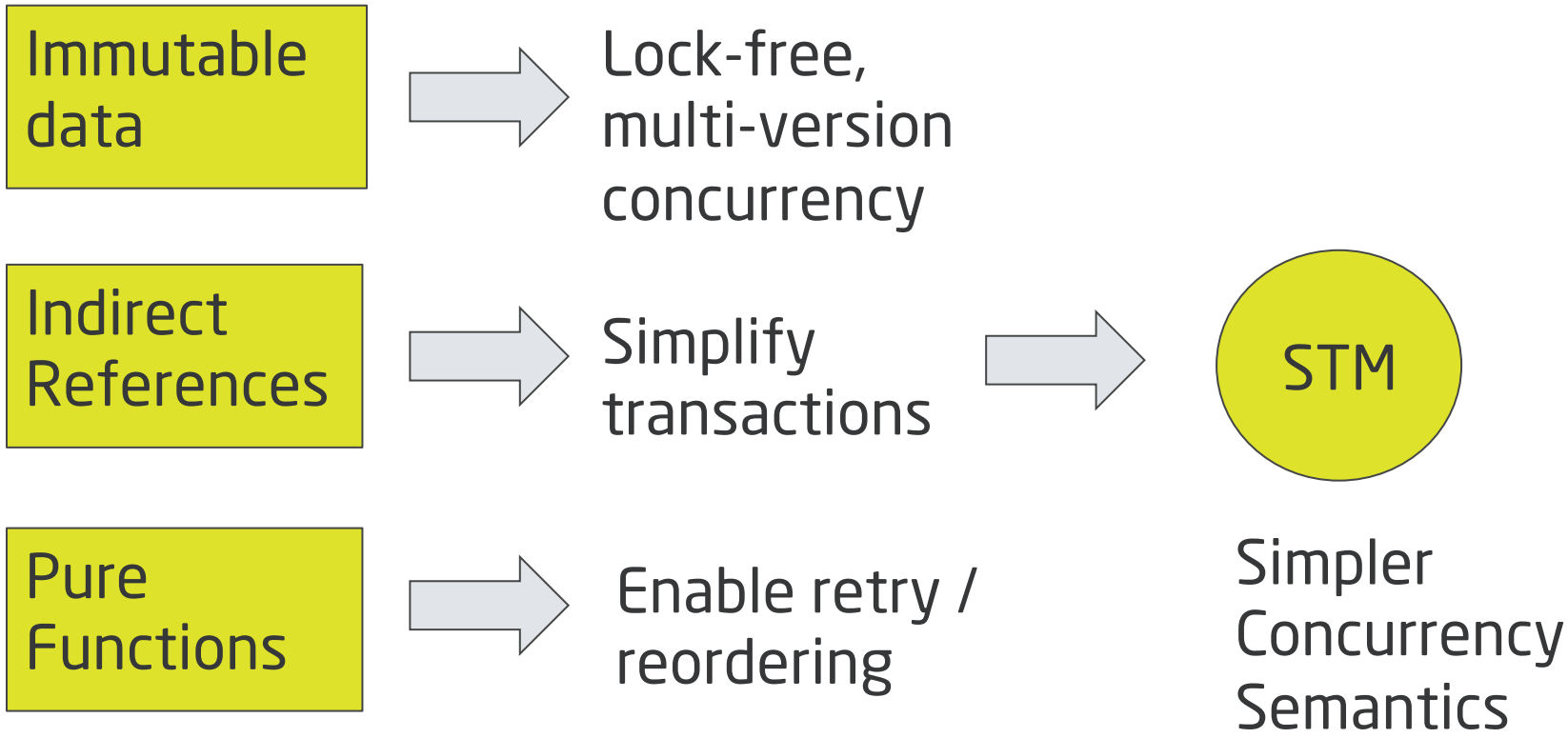

```clojure
(deftest transfer-tests
  (testing "Transfer between accounts"
    (let [a (ref [])
          b (ref [])]
      (transfer a b 10 "message")
      (is (= [{:amount -10, :msg "message"}] @a))
      (is (= [{:amount 10, :msg "message"}] @b)))))
```

file: stm.clj

Software Transactional Memory Conflict Resolution



Concurrency Summary



IT'S ALL ABOUT ABSTRACTIONS

Classes are Islands

```
// C#
class Conference {
  string Name { get; }
  int Year { get; }
}
```

Methods available:

ToString
GetHashCode
Equals
GetType

file: islands.clj

Clojure Data Structures

```
(defrecord Conference [name year])

(def cc (Conference. "Clojure Conj" 2011))
(def cday (Conference. "Community Day" 2012))
(def confs [oredev cday])

confs
=> [{:name "Clojure Conj", :year 2011}
    {:name "Community Day", :year 2012}]

;; key/value map semantics
(:year cday)
=> 2012

(keys cday)
=> (:name :year)
```

file: islands.clj

Clojure Data Structures

```
;; Fields can be added dynamically
(assoc cday :rating :great)

=> {:name ”Community Day",
    :year 2012,
    :rating :great}
```

```
;; A map is a seq of its k/v pairs
(seq cday)

=> ([:name "Community Day"]
    [:year 2012])
```

```
;; Destructuring
(for [[property value] cday]
  (str property ” -> " value))

=> (":name -> Community Day"
    ":year -> 2012")
```

```
confs
=>[{:name "Clojure Conj", :year 2011}
   {:name "Community Day", :year 2012}]
```

file: islands.clj

Functions on Data Structures

```clojure
;; Data works with common functions
(sort-by :name confs)

;; lambda functions
(sort-by (fn [c] (count (:name c))
         confs)
=> ({:name "Clojure Conj", :year 2011}
    {:name "Community Day", :year 2012})

(filter #(= 2012 (:year %)) confs)
=> ({:name "Community Day", :year 2012})
```

```clojure
confs
=>[{:name "Clojure Conj", :year 2011}
   {:name "Community Day", :year 2012}]
```

```javascript
// Javascript (sort-by :name …)

var confs = [
 {name : "Clojure Conj", year : 2011 },
 {name : "Community Day", year : 2012 }];

var sorted = confs.sort(
function(a, b)
{
  if (a.name < b.name)
    return -1
  if (a.name > b.name)
    return 1
  return 0
});
```

file: islands.clj / islands.js

Code to Common Abstractions

Core Abstractions

- Higher-order, first-class fn
- Collections
- Seq
- Records

Core Data Structures

{ :key value }	map
[a b c]	vector
(1 2 3)	list
#{ :a :b :c }	set

Higher-order functions

(`map` *fn* coll)

(`filter` *pred* coll)

(`remove` *pred* coll)

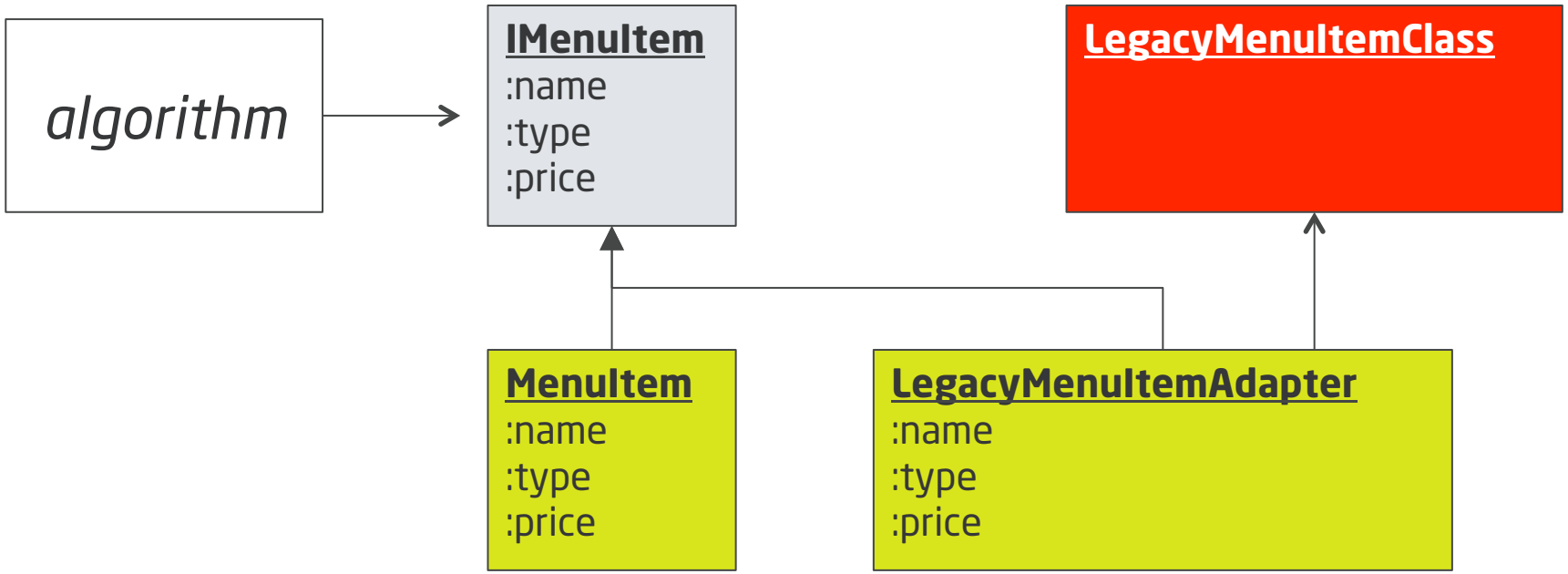
(`sort-by` *fn* coll)

(`group-by` *fn* coll)

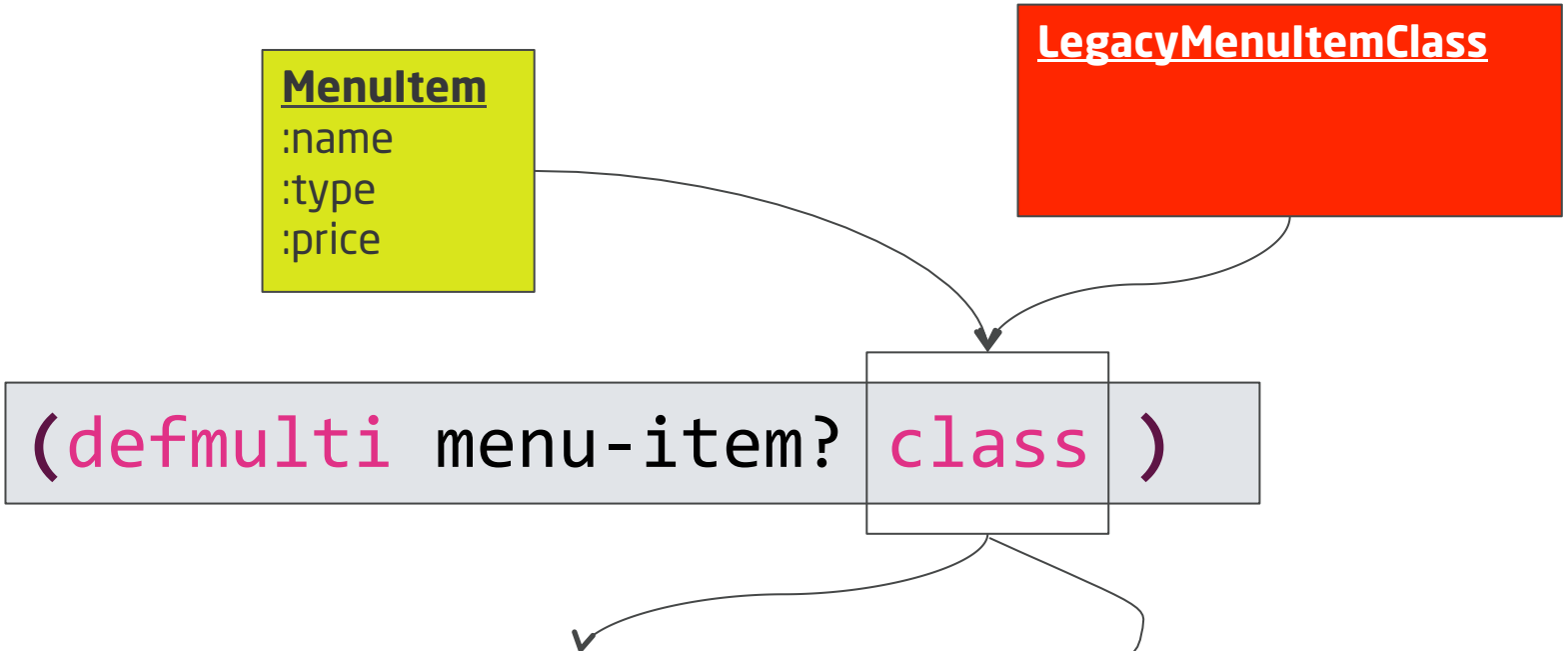
```
// map in C#

// Linq
from x in coll select f(x);

// Pre-Linq
var result = new List…
foreach (var x in coll) {
  result.Add( f(x) );
}

// Extension methods,
// lambda expressions
coll.ConvertAll( x => f(x) );
```

BETTER POLYMORPHISM

Open/Closed Legacy Code : OO



Open/Closed Legacy Code




```
(defmethod menu-item? MenuItem [x] true)
(defmethod menu-item? LegacyMenuItemClass [x] true)
(defmethod menu-item? :default [x] false)
```

file: multimethods.clj

Beyond Static Dispatch

```
{:name "Espresso"
 :type :beverage
 :price 12}

{:name "Big Kahuna Burger"
 :type :food
 :price 100}
```

MenuItem
:name
:type
:price

```
(defmulti description :type )
```

```
(defmethod description :beverage [x]
      (str "Drink a wonderful " (:name x)))
(defmethod description :food [x]
      (str "Savour a tasty " (:name x)))
```

file: multimethods.clj

SPECIALIZING THE IMPLEMENTATION LANGUAGE

How would you add an *unless* keyword to C#?

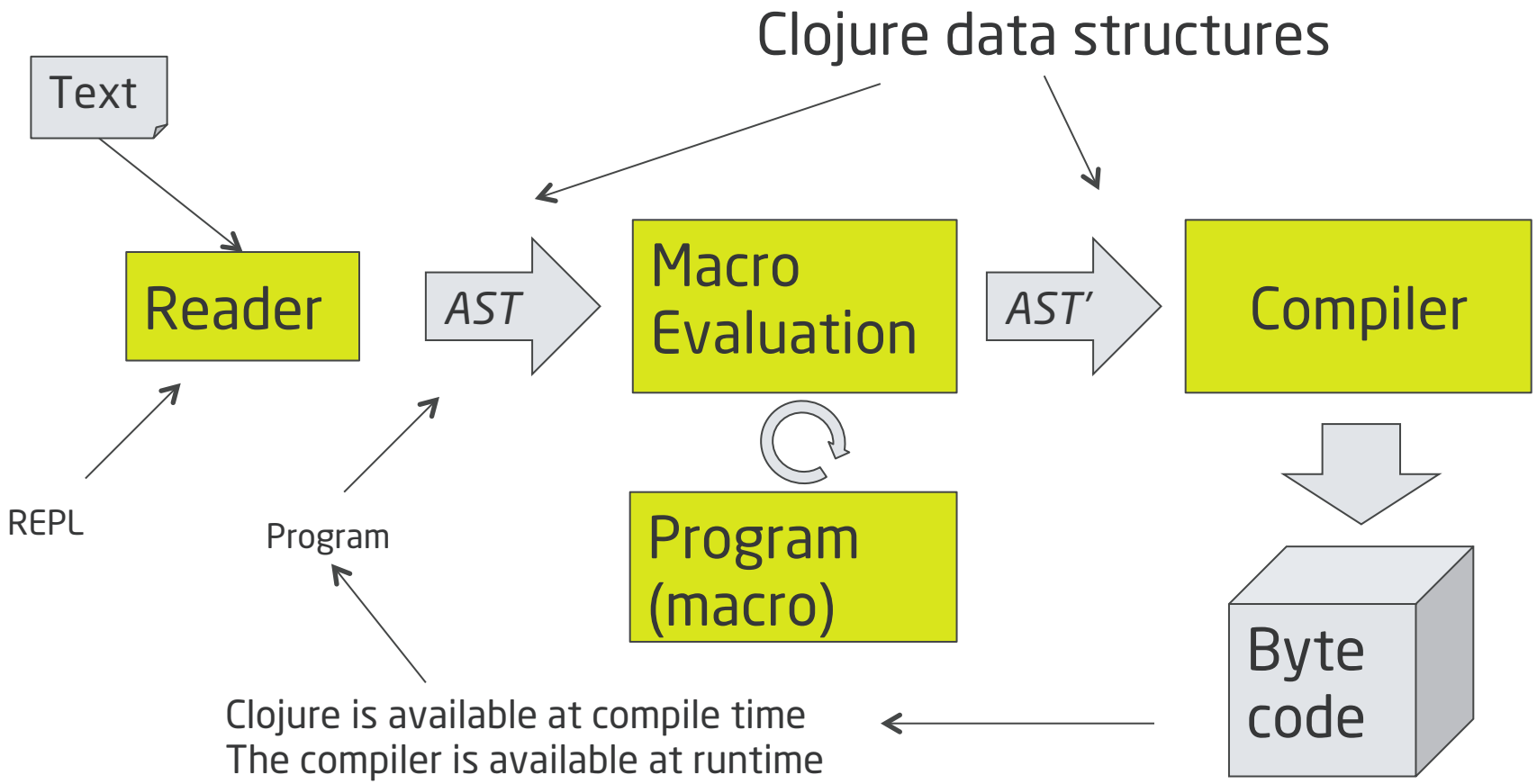
```
public WeakSetPerson(Person p)
{
    this.person = p unless (p == null);
}
```

How would you build Active Record?

```
class Manager < ActiveRecord::Base
  has_one :department
end
```

```
class Module
 def my_attr(symbol)
  class_eval "def #{symbol}; @#{symbol}; end"
  class_eval "def #{symbol}=(value); @#{symbol} = value; end"
 end
end
```

The Clojure Compilation Pipeline



The whole language always available*

- Homoiconic
  - A program is a data structure (AST)
  - "Code is data is code"
- A **macro** is a function that transforms the program data at compile-time
- Functions are data structures, too.
- Clojure at compile-time, Clojure at runtime.

** Paul Graham, What Made Lisp Different, 2002*

Adding “unless” to Clojure

```
(defmacro unless
  [test & body]
  (list 'if test nil (cons 'do body)))
```

```
(macroexpand-1 '(unless (neg? x)
                        (println "x is non-neg")))

;; expands to
(if (neg? x)
    nil
    (do (println "x is non-neg")))
```

Read

Macro eval

Compile

* Actually, this is the Clojure when-not macro

CONCLUSIONS

Small, Powerful and Extensible

Simple core language

Excellent **function library**

Persistent collections

Immutability by default

Sane Reasoning about State

Software Transactional Memory

Sane Concurrency

Code is data: **powerful meta programming** features

Typed Clojure Gradual typing	**core.logic** Logic programming	**Trammel** Code contracts	**Eastwood** C Lint

Simple libraries make Haskell, Prolog and Eiffel hackers feel at home

Reducing the Complexity of the Implementation Domain

Problem	Simplification
Spaghetti code	Structured programming, OO
Memory management	Garbage collection
Side-effects	**Pure functions**
Sharing data	Message passing, value semantics **Immutable data**
Concurrency / locks	**Software Transactional Memory** Message based concurrency Offline lock patterns, ...
Composability	**Common abstractions**, higher-order functions
Limitations of implementation language	**Macros** DSLs, Design patterns

Where to go from here

IDEs

- Emacs SLIME
- Clojurebox (Emacs)
- Eclipse "Counter clockwise"
- NetBeans "Enclojure"
- Intelli/J "La Clojure"
- Visual Studio "vsClojure"

Online REPL

www.tryclj.com

Tools

- Leiningen package management, build ... http://leiningen.org/
- www.clojure.org

Thank you

Husk at udfylde evalueringsskema:

http://bit.ly/cd**2013**b3

Download the slides and examples:

https://github.com/mjul/clojure-communityday-2012

Martin Jul
martin@mjul.com
@mjul

Work
mj@ative.dk
@ativedk
http://www.ative.dk

http://bit.ly/cd**2013**b3