Top Ten Things to Learn from Clojure that will make you a better developer in any language

Cool Languages track
Øredev 2011, Malmö, 11/11/11 11:11


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

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"

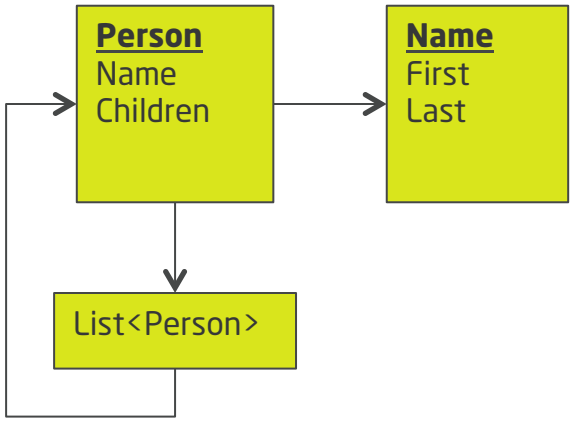
Reducing the Complexity of the Implementation Domain

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

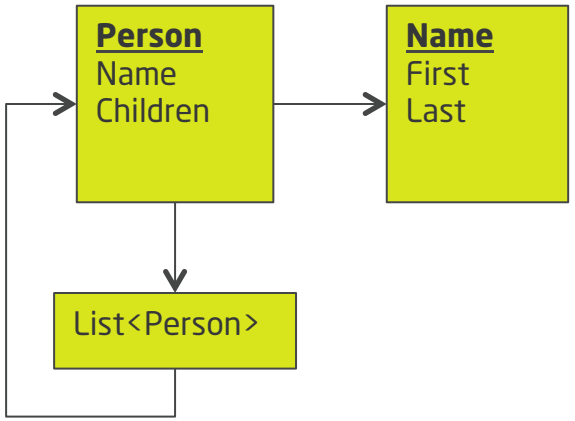
Mutable state: What is wrong with this code?

```
// 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);
```

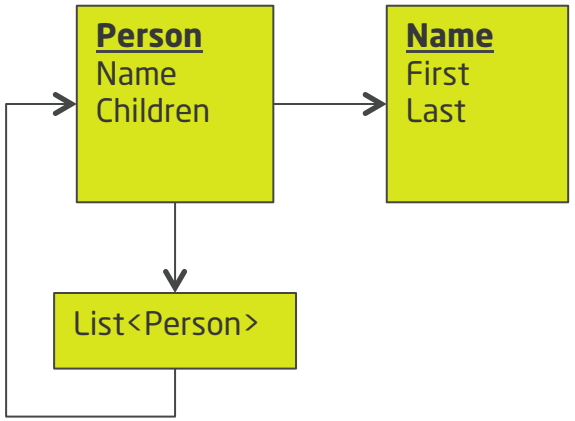



```
alpha.Name.Last = "Omega";
alpha.Children.Add(new Person(new Name("Gamma", "Alphadaughter")));
DoSomethingTo(alpha, beta);
```

What is the state after this?

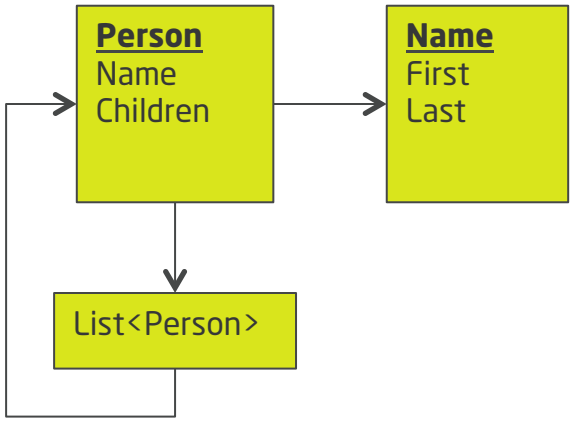
Mutable state:
What is wrong with this code?

```
// 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();
        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

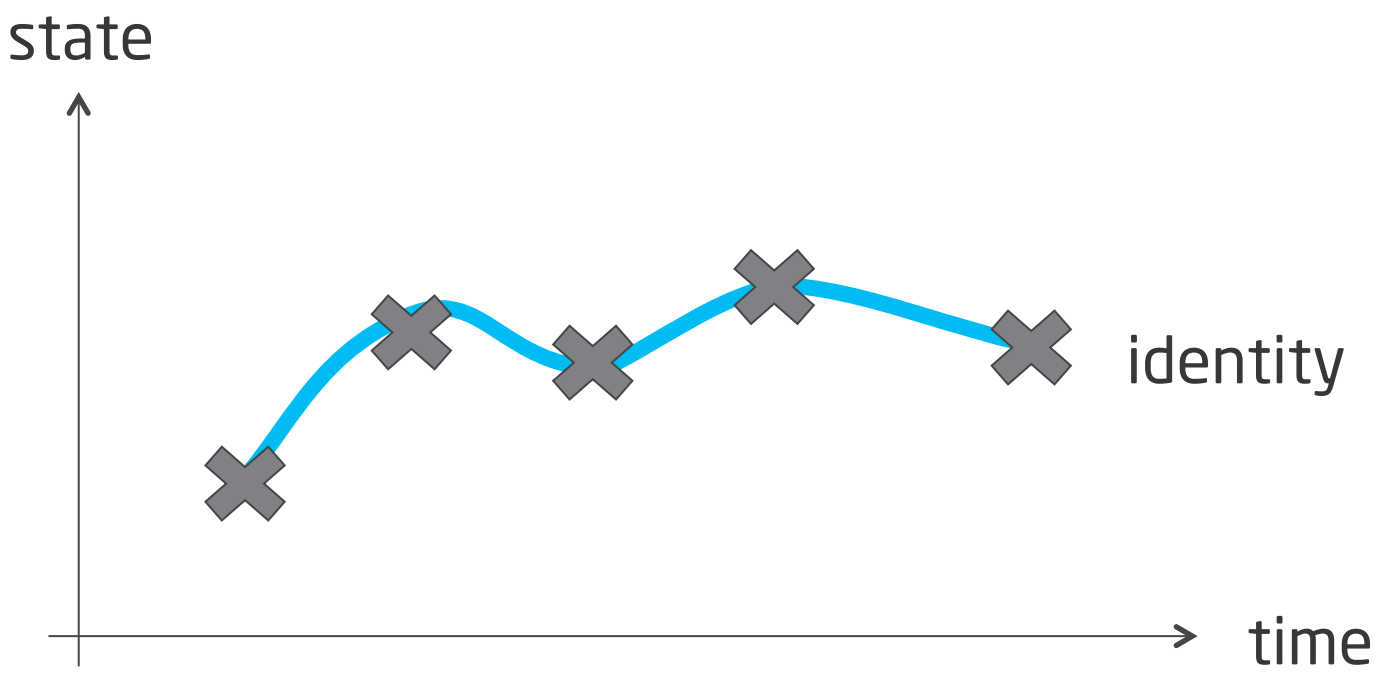
- **Encapsulation** is hard
  - clone in, clone out
- **Ownership** is hard
  - “Entities” and “Value Objects”
- **Reasoning** about state is hard
- **Concurrency** is even worse



Maybe it’s time to stop

IMMUTABILITY

Philosophy of State and Identity



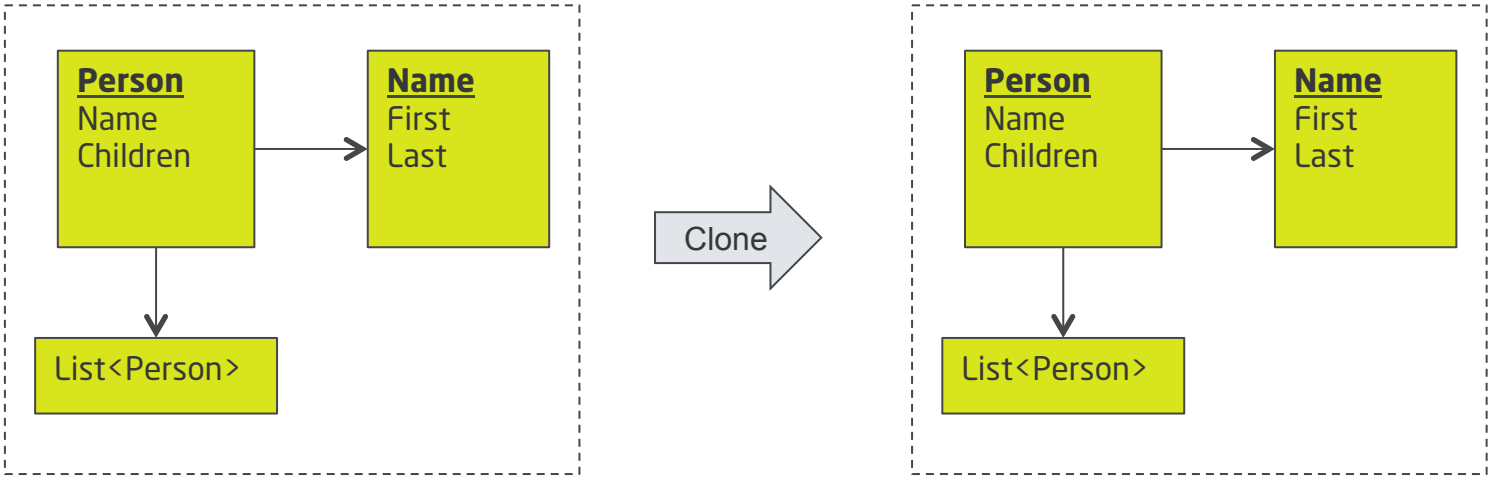
Advantages of Immutability

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

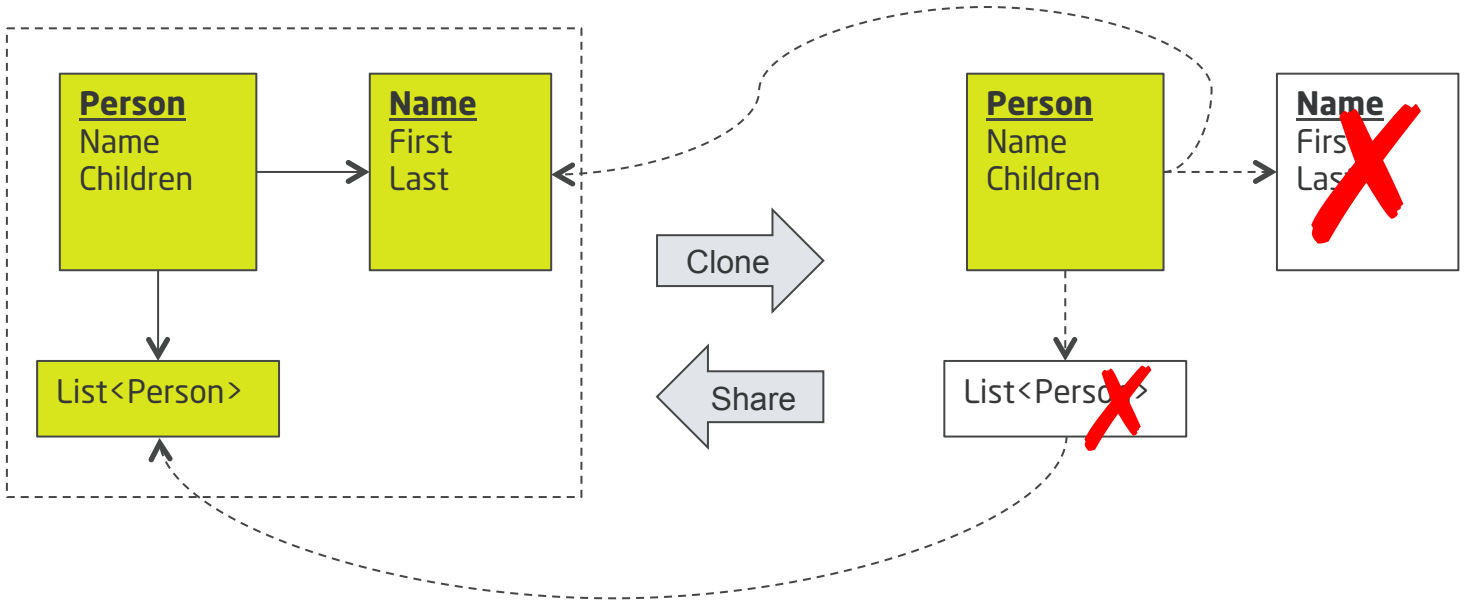
Disadvantages of Immutability

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

Structural Sharing

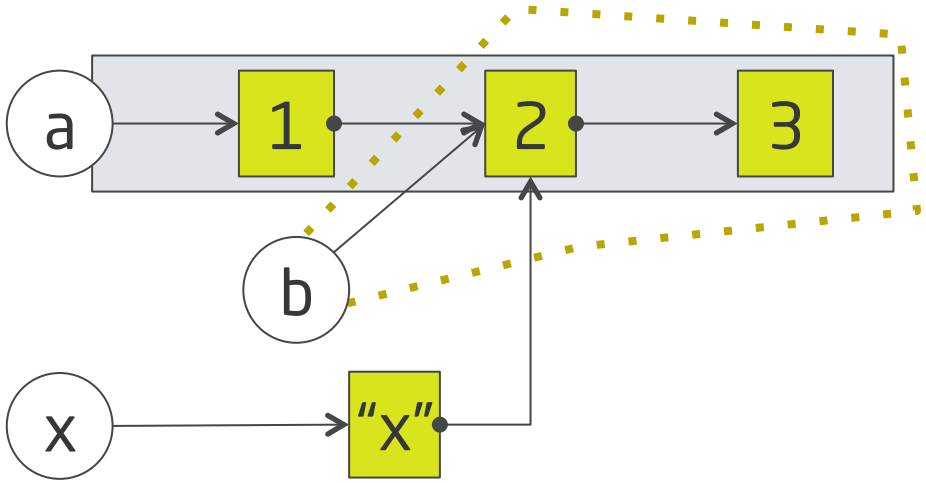


Structural Sharing



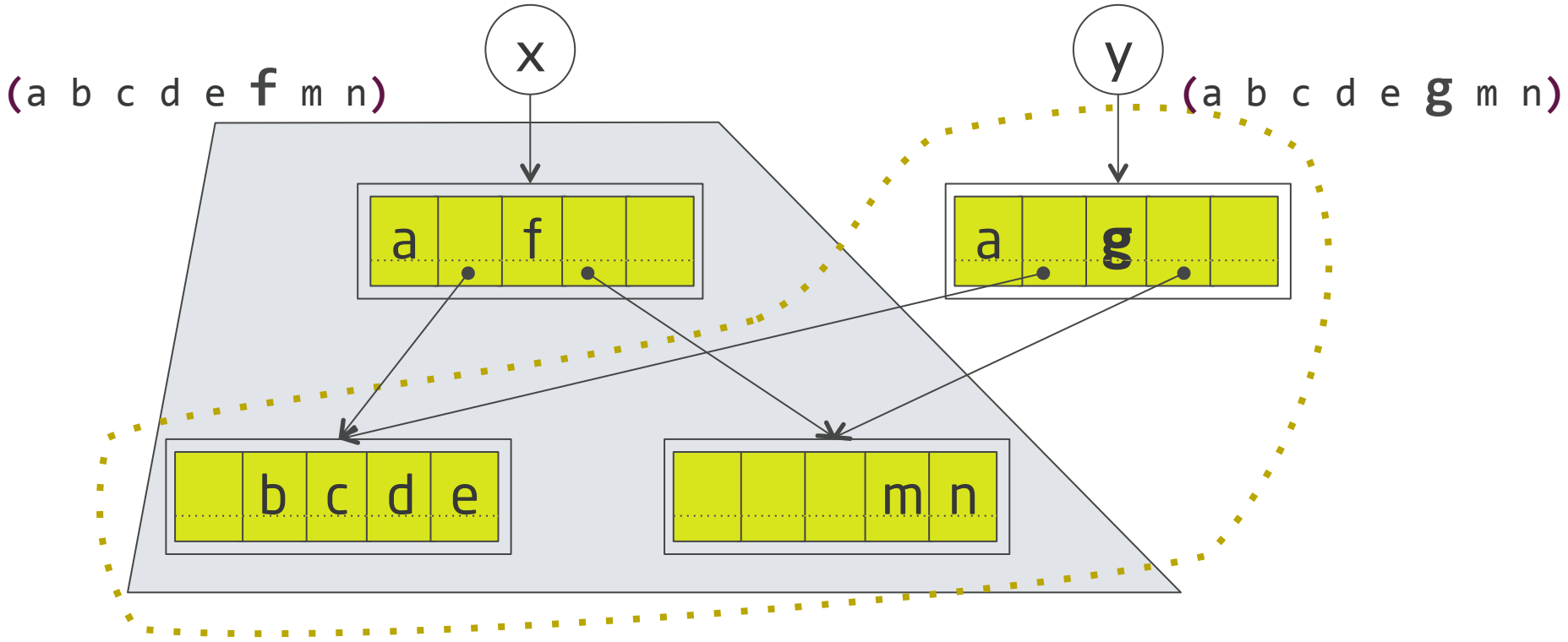
Persistent Collections for performance

```
(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

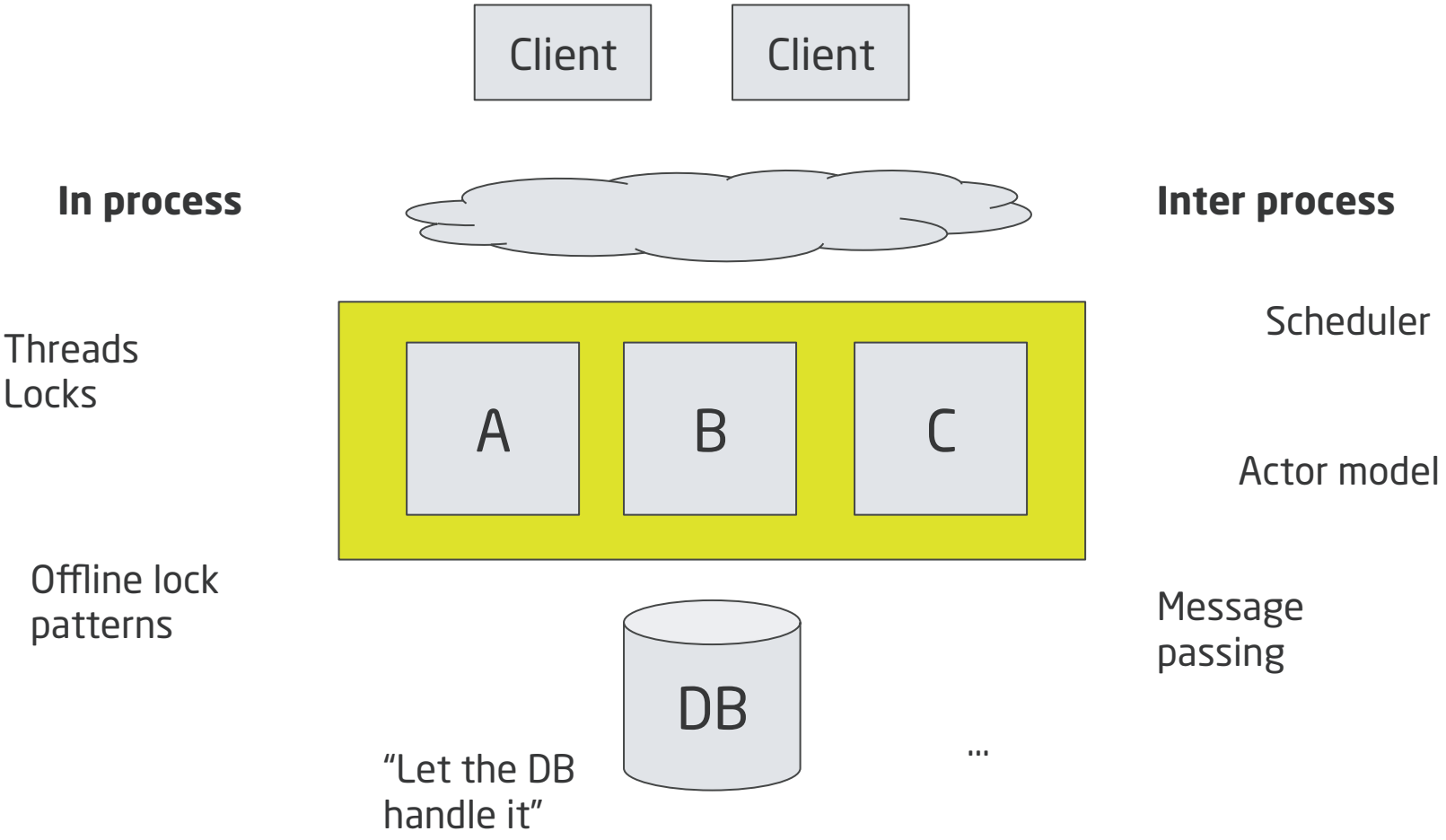
Persistent Collections implemented with hash tries



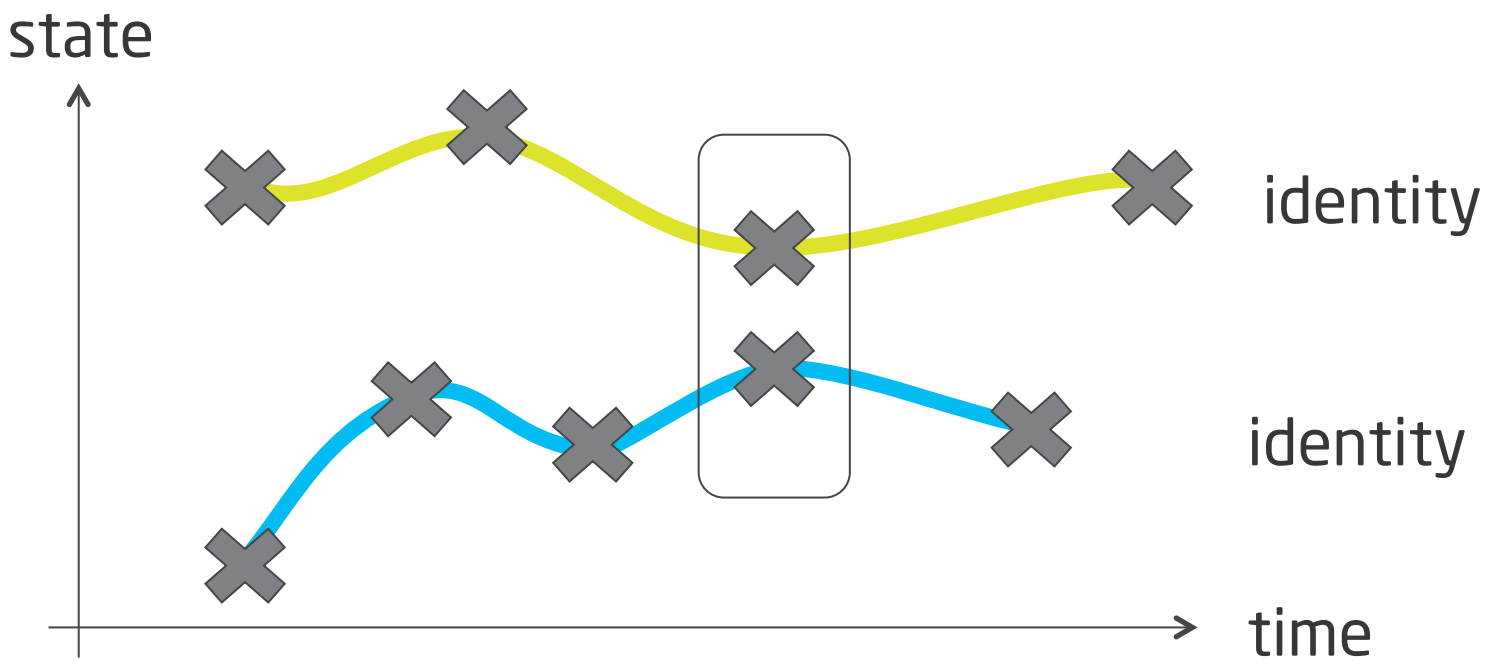
Extremely simplified diagram!
For full details see: Fast and Space Efficient Trie Searches, Bagwell [2000]

CONCURRENCY WITH SOFTWARE TRANSACTIONAL MEMORY

Concurrency Strategies

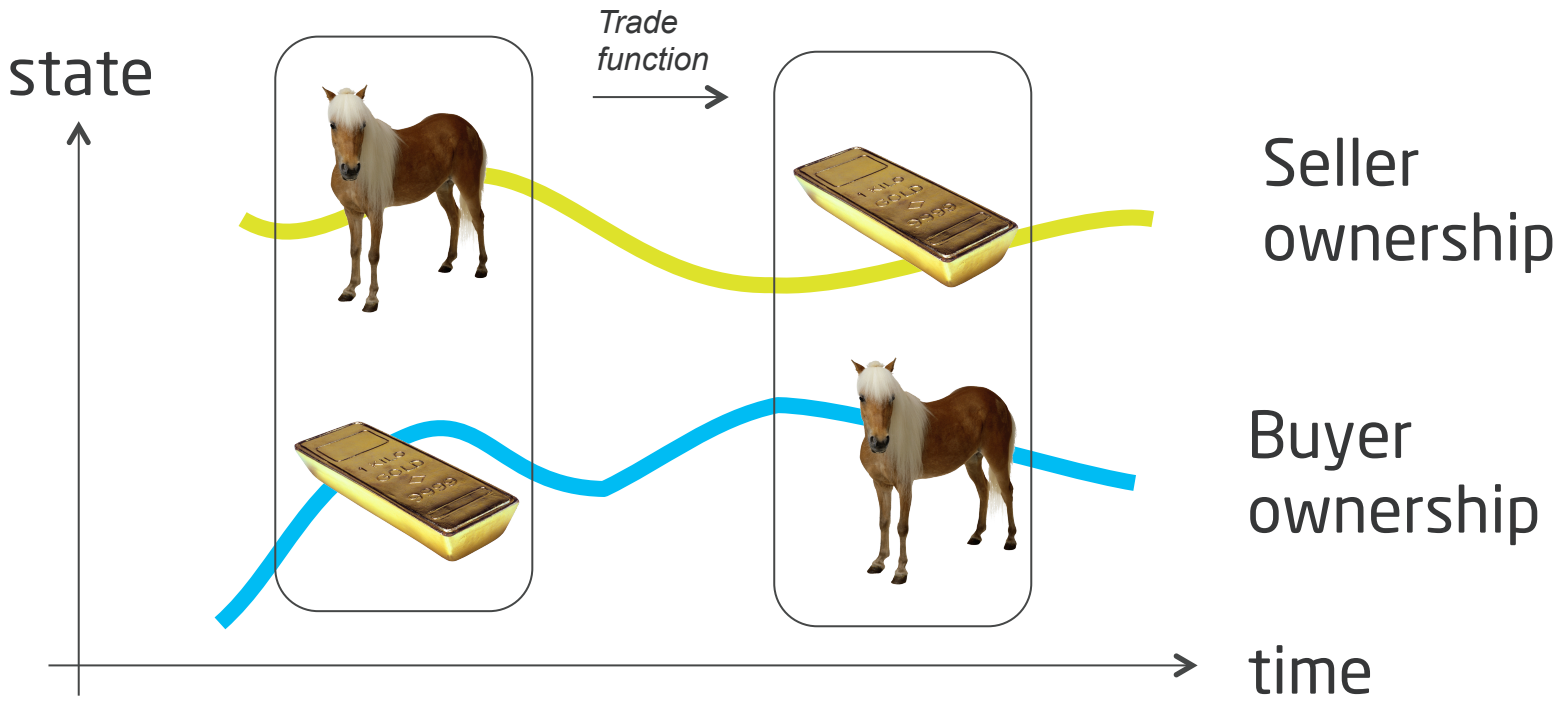


Clojure Concurrency

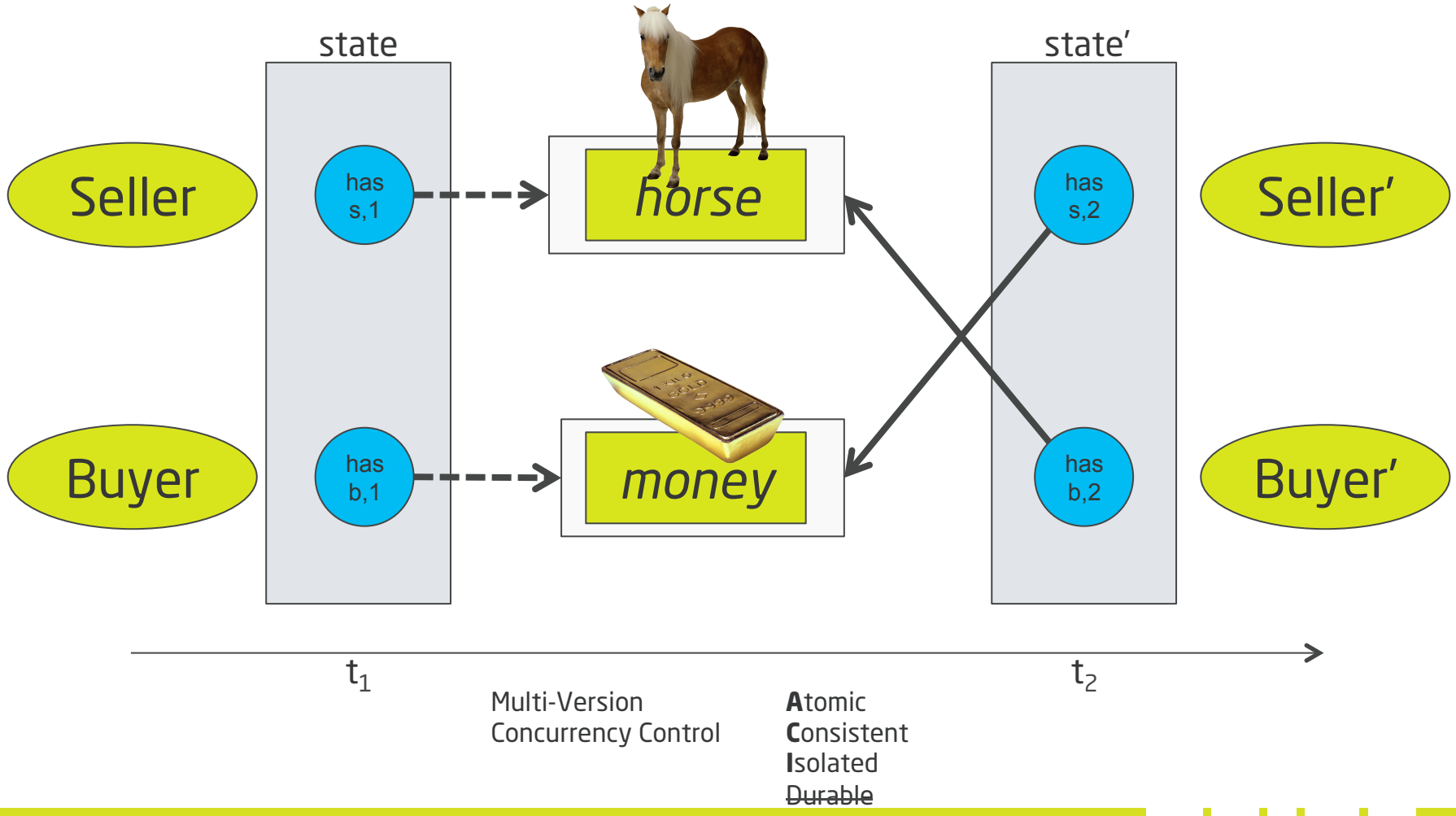


- **Indirect** references to immutable data structures
- Concurrency semantics for references
  - Automatic/enforced
  - No locks

Clojure Concurrency



Software Transactional Memory

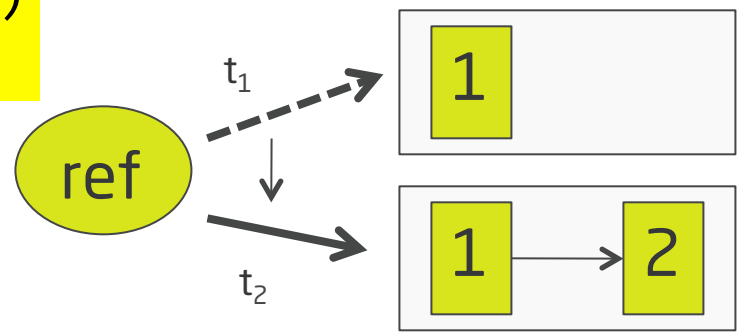


STM Example

```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)))
```

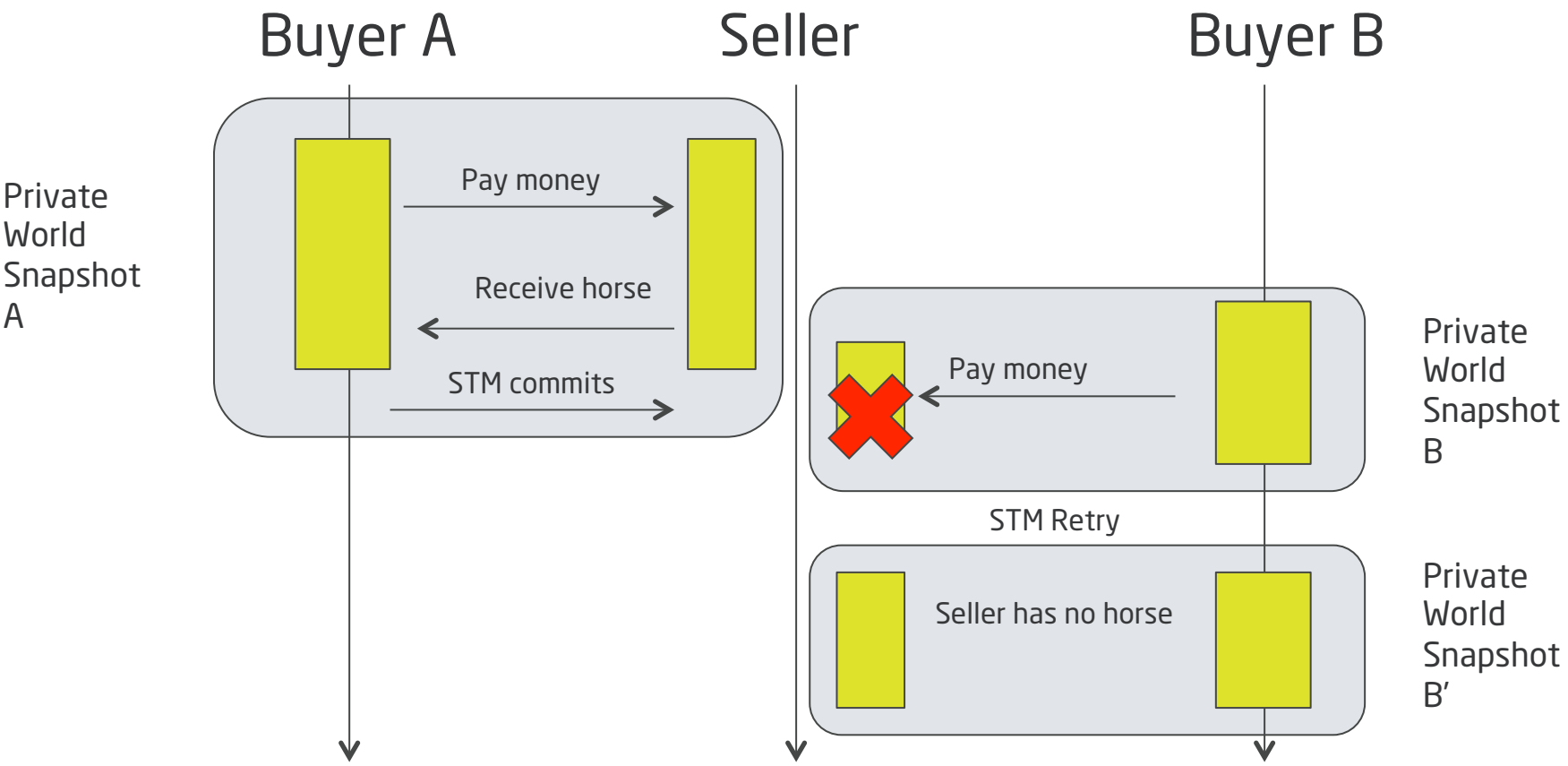
:amount	:msg
1000	Initial balance
-170	Train fare
-40	Coffee

```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)))))
```

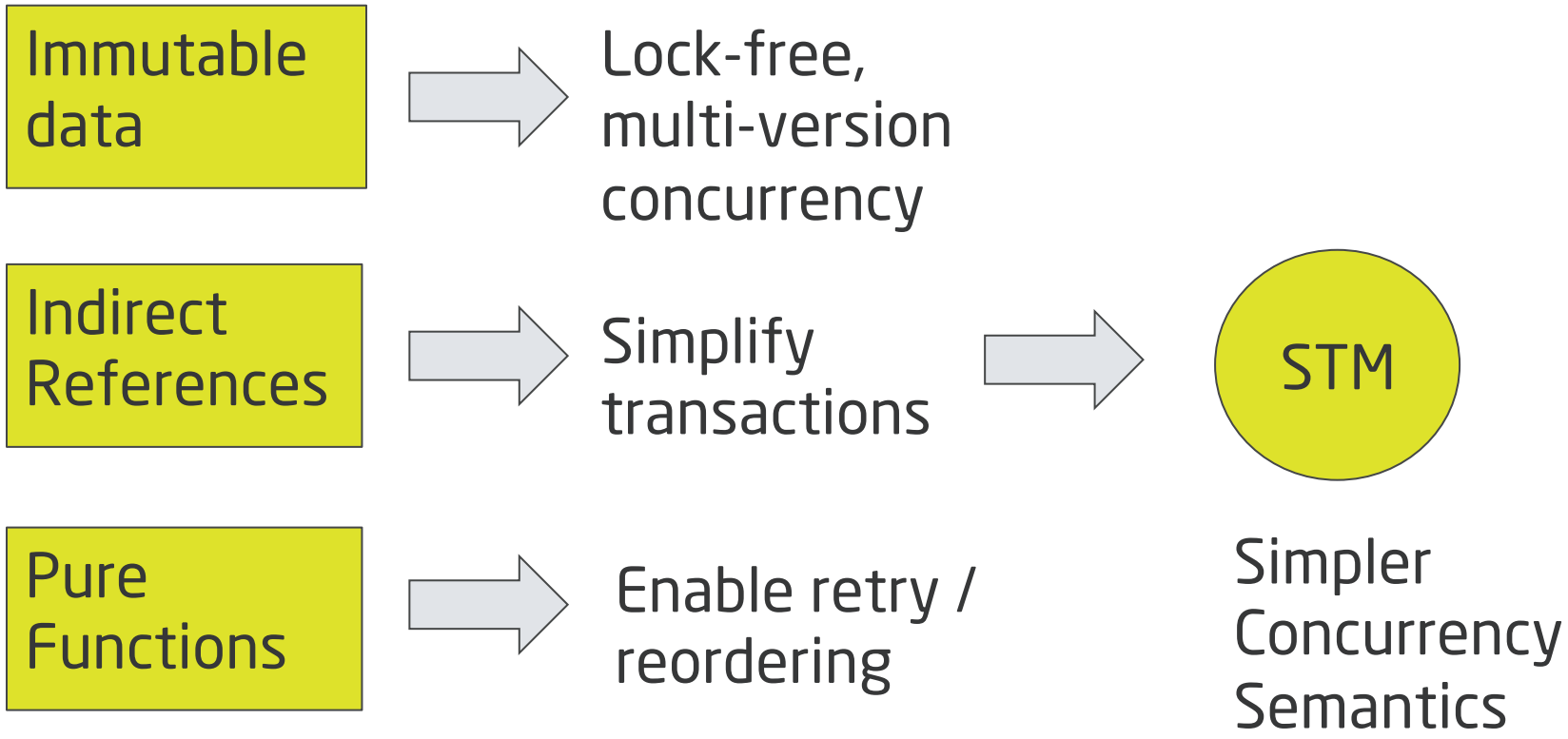


file: stm.clj

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 oredev (Conference. "Øredev" 2011))
(def cc (Conference. "Clojure Conj" 2011))
(def confs [oredev cc])

confs
=>[{:name "Øredev", :year 2011}
   {:name "Clojure Conj", :year 2011}]

;; key/value map semantics
(:year oredev)
=> 2011

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

file: islands.clj

Clojure Data Structures

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

;; lambda functions
(sort-by (fn [c] (count (:name c))
         confs)

(filter #(= 2011 (:year %)) confs)
```

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

=> {:name "Øredev",
    :year 2011,
    :rating :great}

;; It is a seq of its k/v pairs
(seq oredev)

;; Destructuring
(doseq [[property value] oredev]
  (println property "->" value))
```

```clojure
;; confs
[{:name "Øredev", :year 2011}
 {:name "Clojure Conj", :year 2011}]
```

file: islands.clj

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

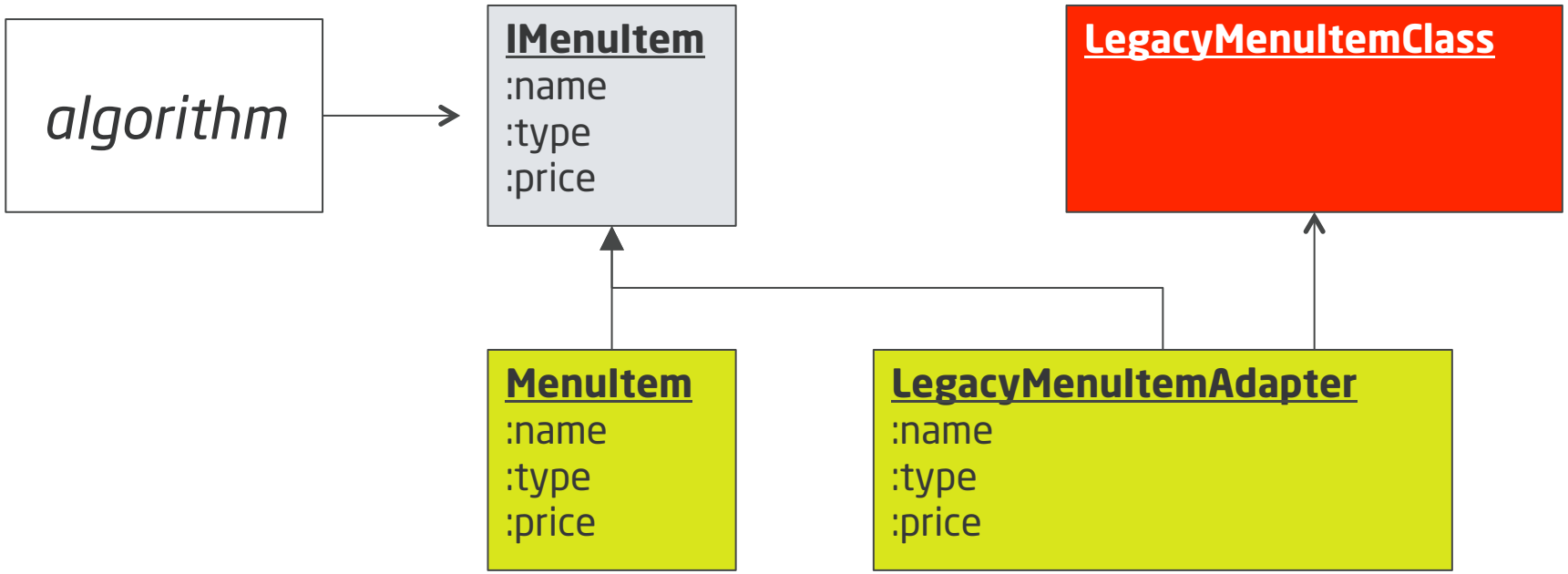
Higher-order functions

(map *fn* coll)
(filter *pred* coll)
(remove *pred* coll)
(sort-by *fn* coll)
(group-by *fn* coll)

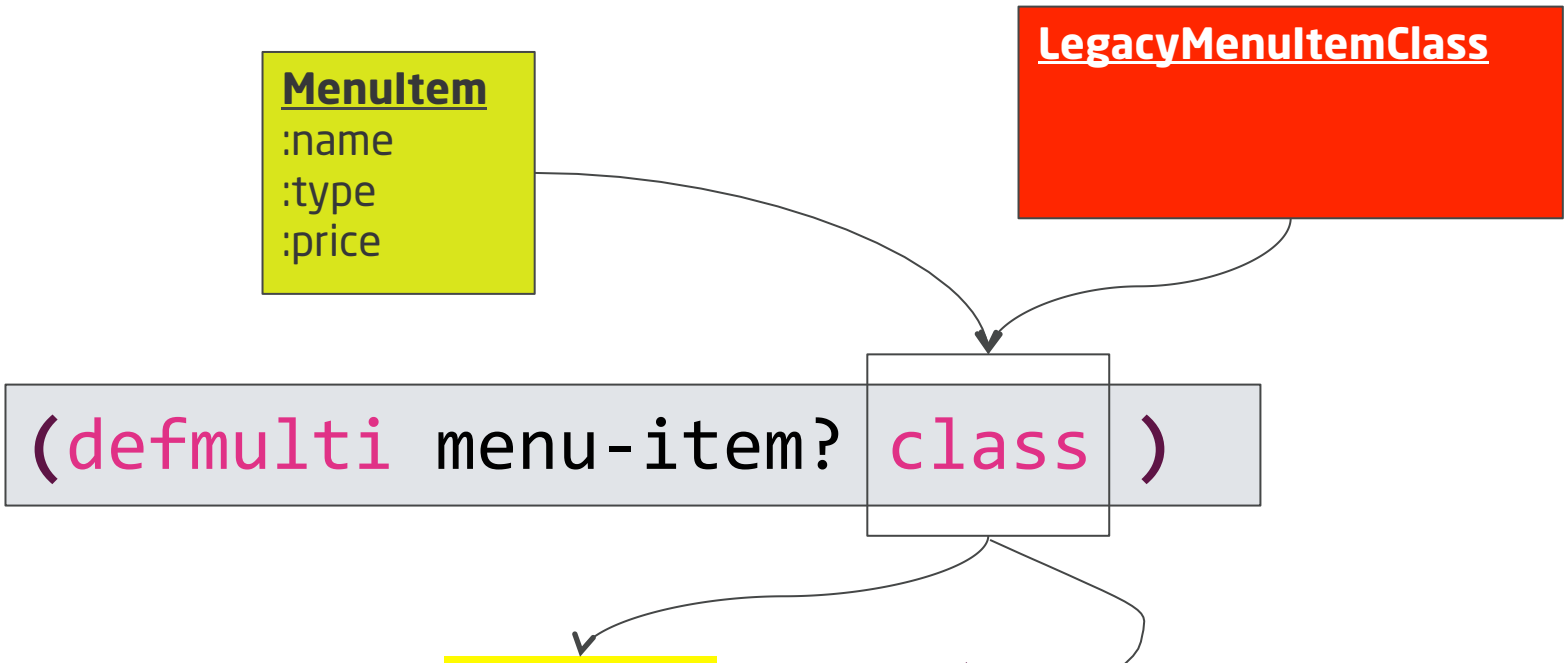
```
// Linq
from x in coll select f(x);

// 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

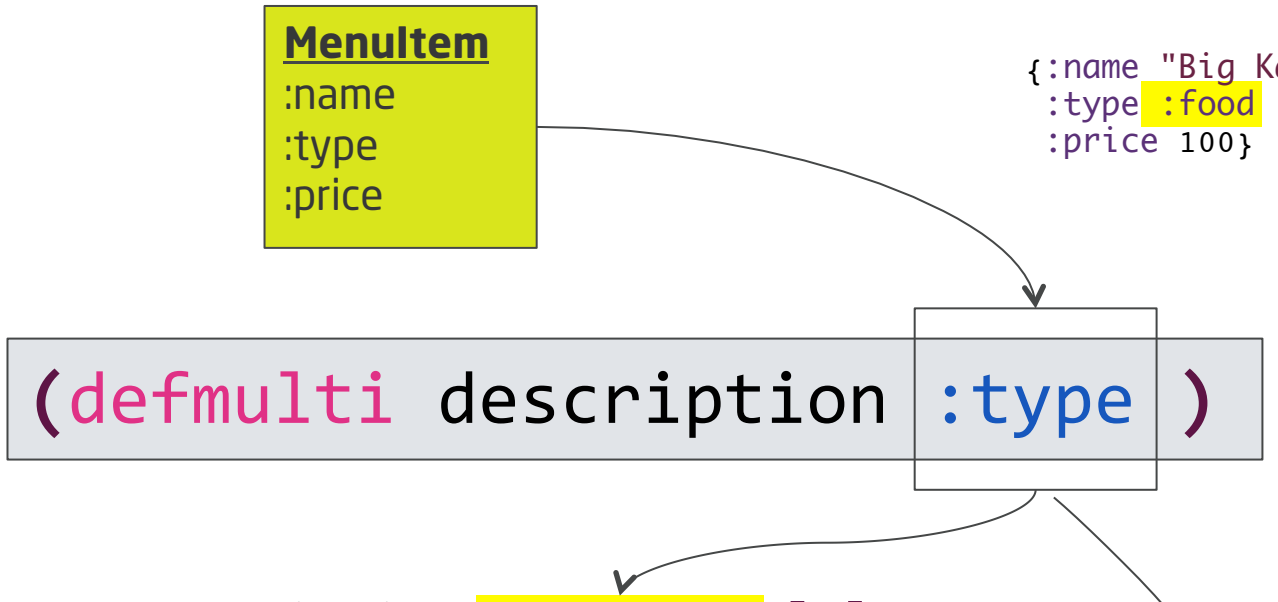



```
(defmethod menu-item? MenuItem [x] true)
(defmethod menu-item? LegacyMenuItemClass [x] true)
(defmethod menu-item? :default [x] false)
```

file: multimethods.clj

Beyond Static Dispatch

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

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




```
(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#?

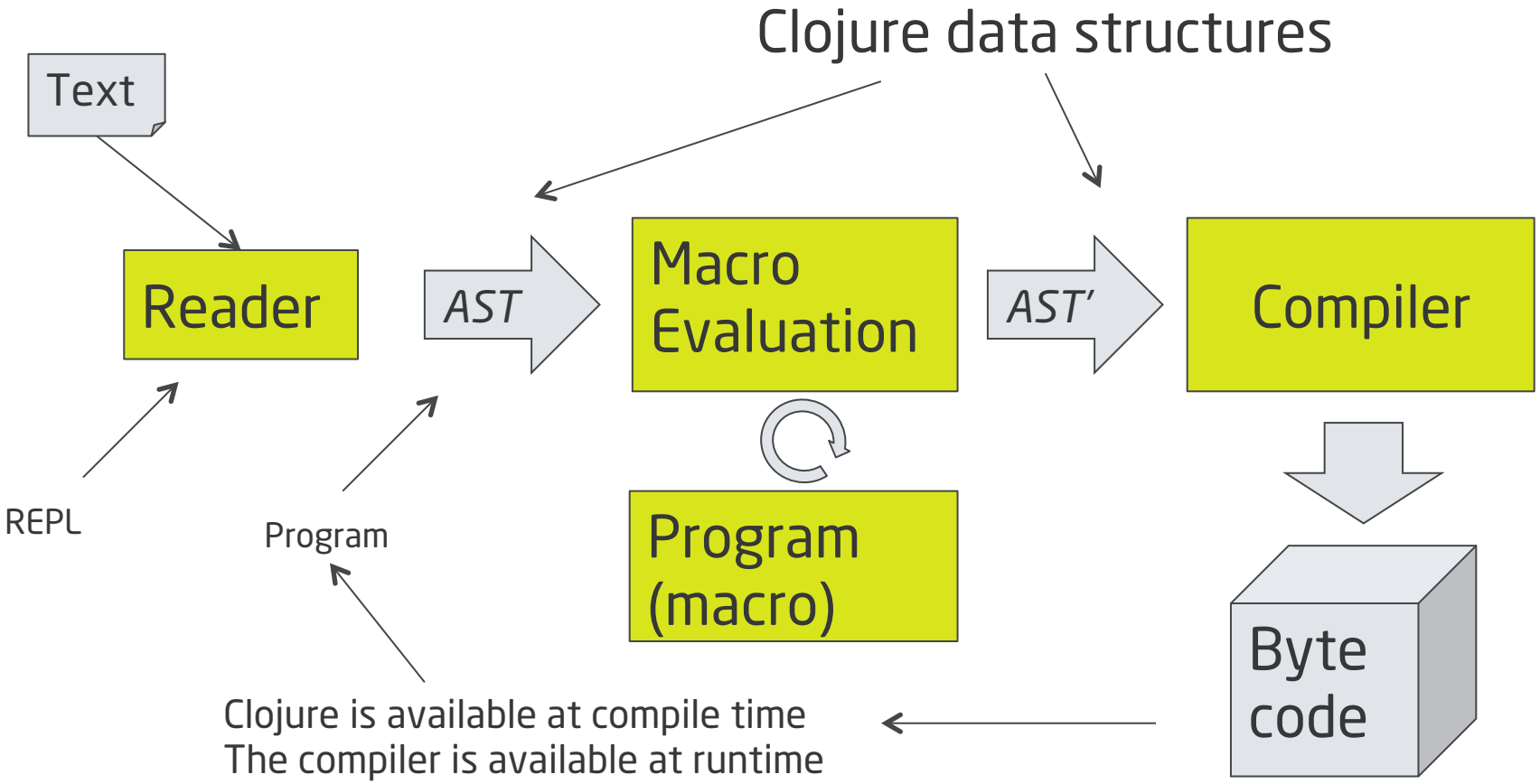
```
public WeakSetPerson(Person p)
{
    this.person = p unless (p == null);
}
```

How would you build Active Record?

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

```ruby
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*

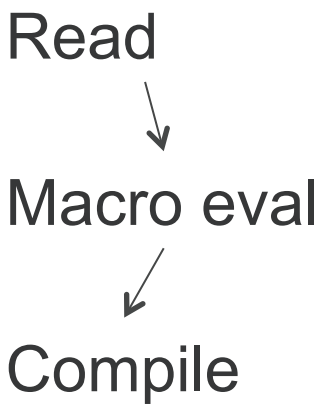
- Homoiconic
  - A program is a data structure (AST)
  - "Code is data is code"
- A **macro** is a function that transforms the program data at compile-time
- Functions are data structures, too.
- Clojure at compile-time, Clojure at runtime.

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

Adding “unless” to Clojure

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

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

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



* Actually, this is the Clojure when-not macro

CONCLUSIONS

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

Top 10 Things…

1. Default to immutability
2. Write pure functions
3. Use structural sharing
4. Minimize the scope of mutation
5. You don't need locks
6. Use common abstractions for composability
7. Dependency inversion principle goes far
8. Code is Data
9. Not everything is an object
10. Polymorphism can go much further

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

- Cake (build, test +)
- Leiningen (ditto)

- www.clojure.org

Thank you

Download the slides and examples here:
https://github.com/mjul/top-10-clojure-oredev-2011

Martin Jul
martin@mjul.com
Twitter: @mjul

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

Code
https://github.com/mjul
https://github.com/ative