

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

Martin Jul (mj@ative.dk / @mjul)
NDC 2011, Oslo, June 10, 2011

Why Clojure?

- Small
- Powerful
- Elegant
- Functional
- Extensible
- Concurrency
- Interoperable

Reducing Complexity of the Implementation Domain

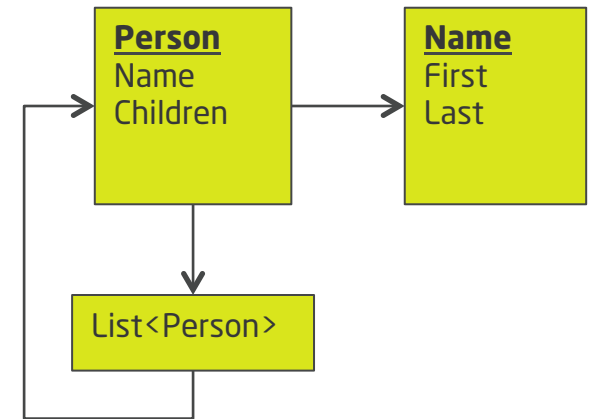
Problem	Simplification
Spaghetti code	Structured programming, OO
Memory management	Garbage collection
Side-effects	Pure functions
Sharing data	Message passing, value semantics Immutable data
Concurrency / locks	Software Transactional Memory Message based concurrency Offline lock patterns, ...
Composability	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?

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

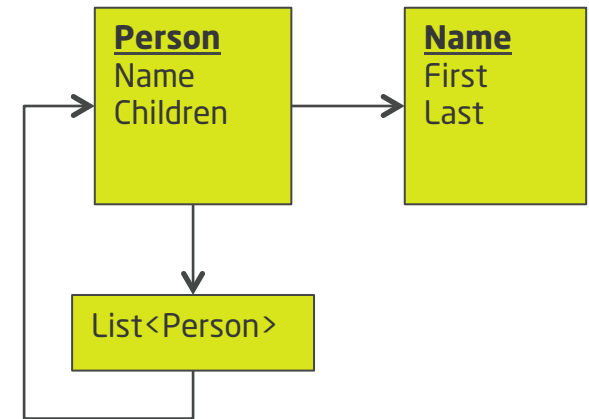
public class Person {
    public Person(Name name, List<Person> children)
    {
        this.Name = name;
        this.Children = children;
    }
    public Name Name { get; set; }
    public List<Person> Children { get; }
}
```



Mutable state: What's wrong with this code?

What is the state after this?

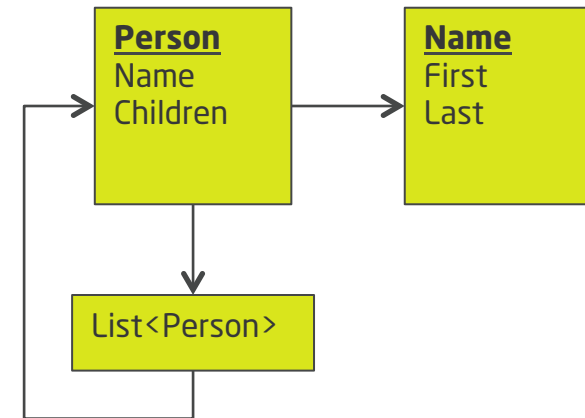
```
var noChildren = new List<Person>();  
var alpha = new Person(new Name("Alpha", "Sister"), noChildren);  
var beta  = new Person(new Name("Beta", "Sister"), noChildren);  
  
alpha.Name.Last = "Omega";  
alpha.Children.Add(new Person(new Name("Gamma", "Sisterdaughter")));  
DoSomethingTo(alpha, beta);
```



Mutable state: What is wrong with this code?

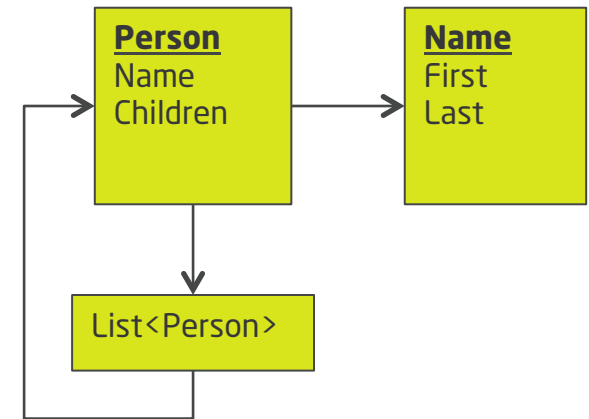
```
public class Name {
    public String First { get; set; }
    public String Last { get; set; }
}

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 p) { this.children.Add(p.DeepClone()); }
}
```



Mutable state

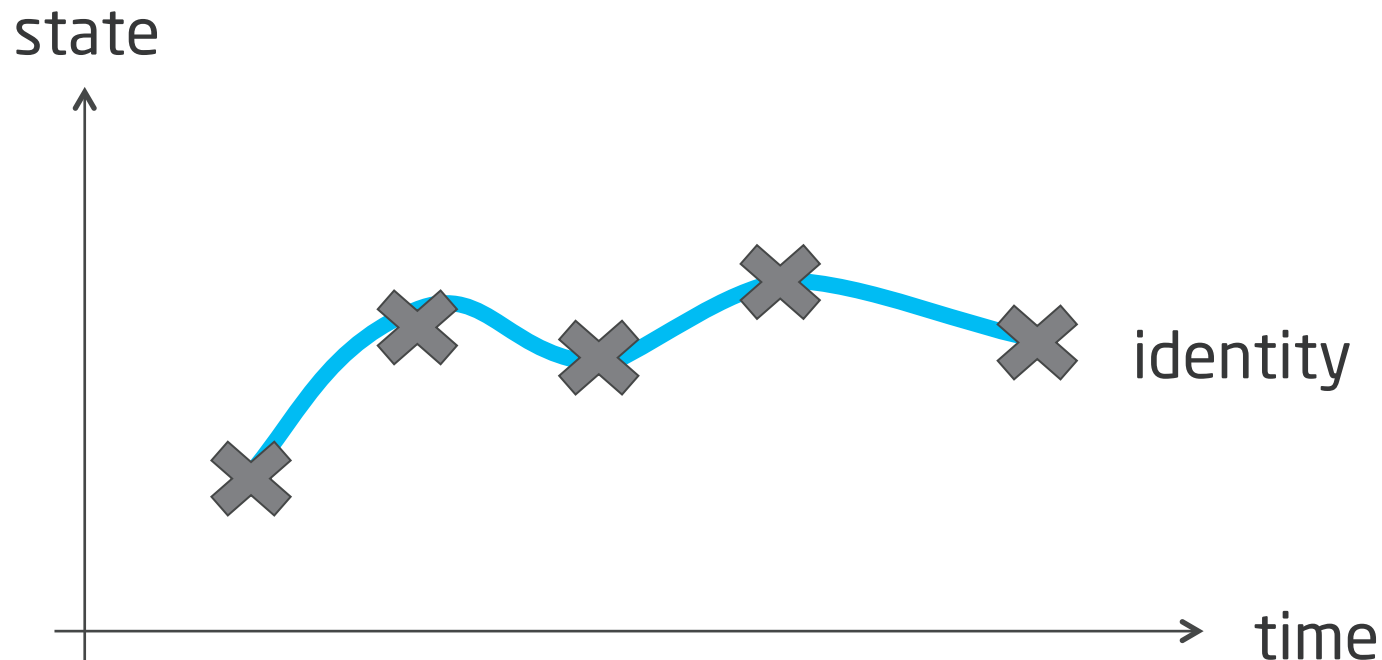
- **Encapsulation** is hard
 - clone in, clone out
- **Ownership** is hard
 - “Entities” and “Value Objects”
- **Reasoning** about state is hard
- **Concurrency** is even worse



Maybe it's time to stop

IMMUTABILITY

Philosophy of State and Identity



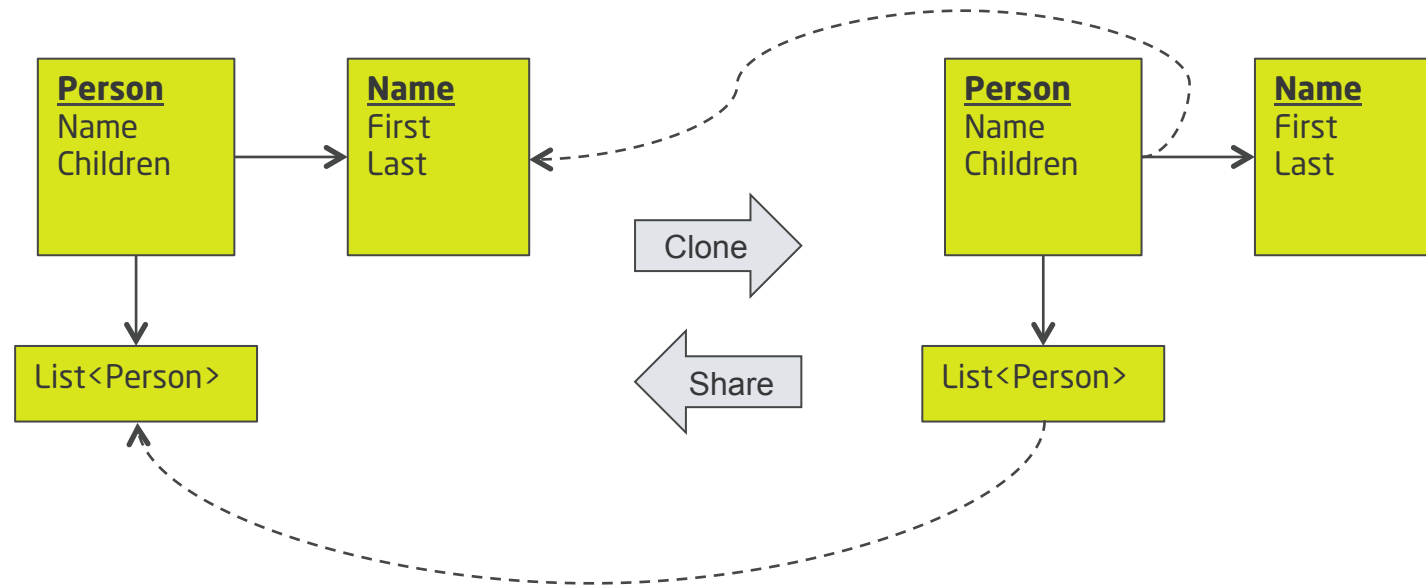
Advantages of Immutability

- 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

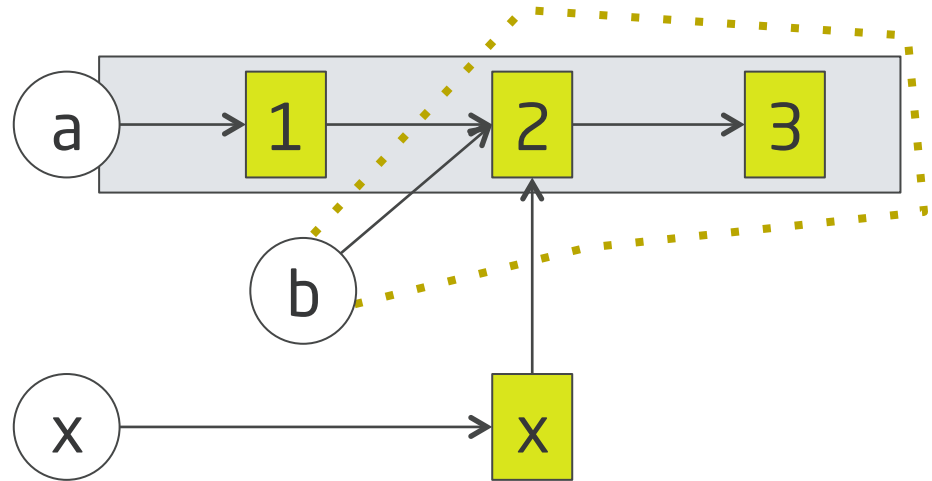


Persistent Collections for performance

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

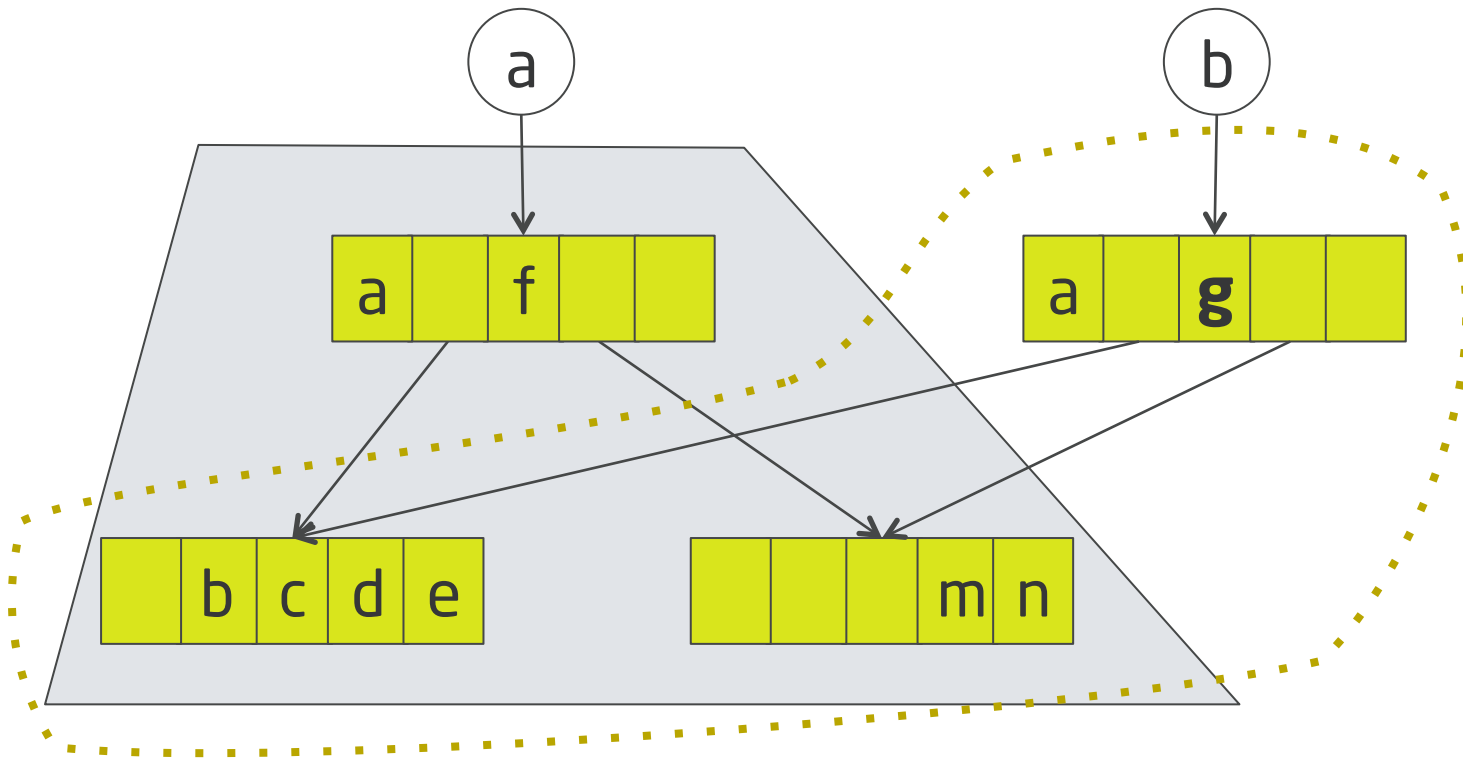
```
(def b (rest a))  
=> (2 3)
```

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



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

Persistent Collections implemented with hash tries

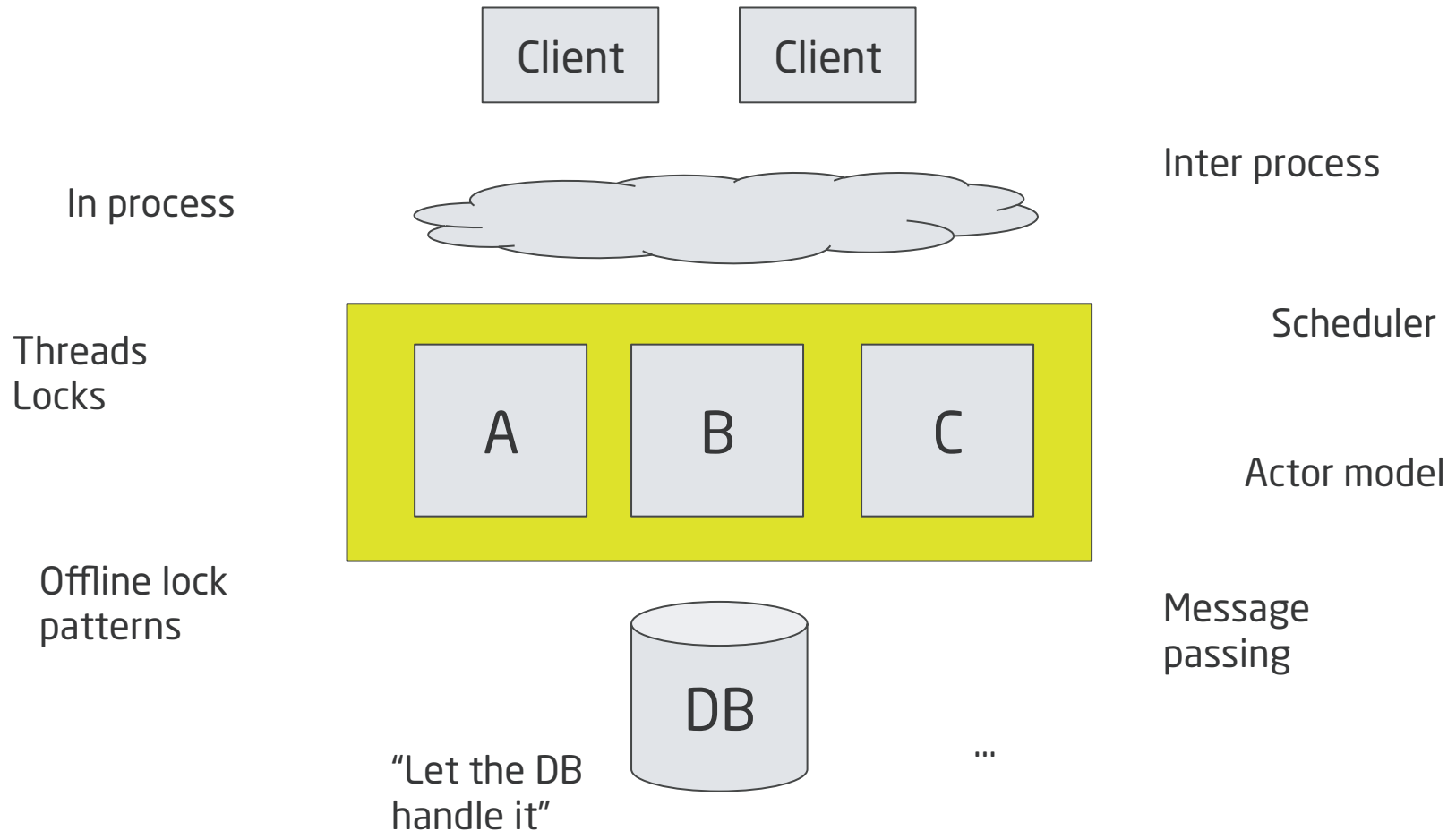


Extremely simplified diagram!

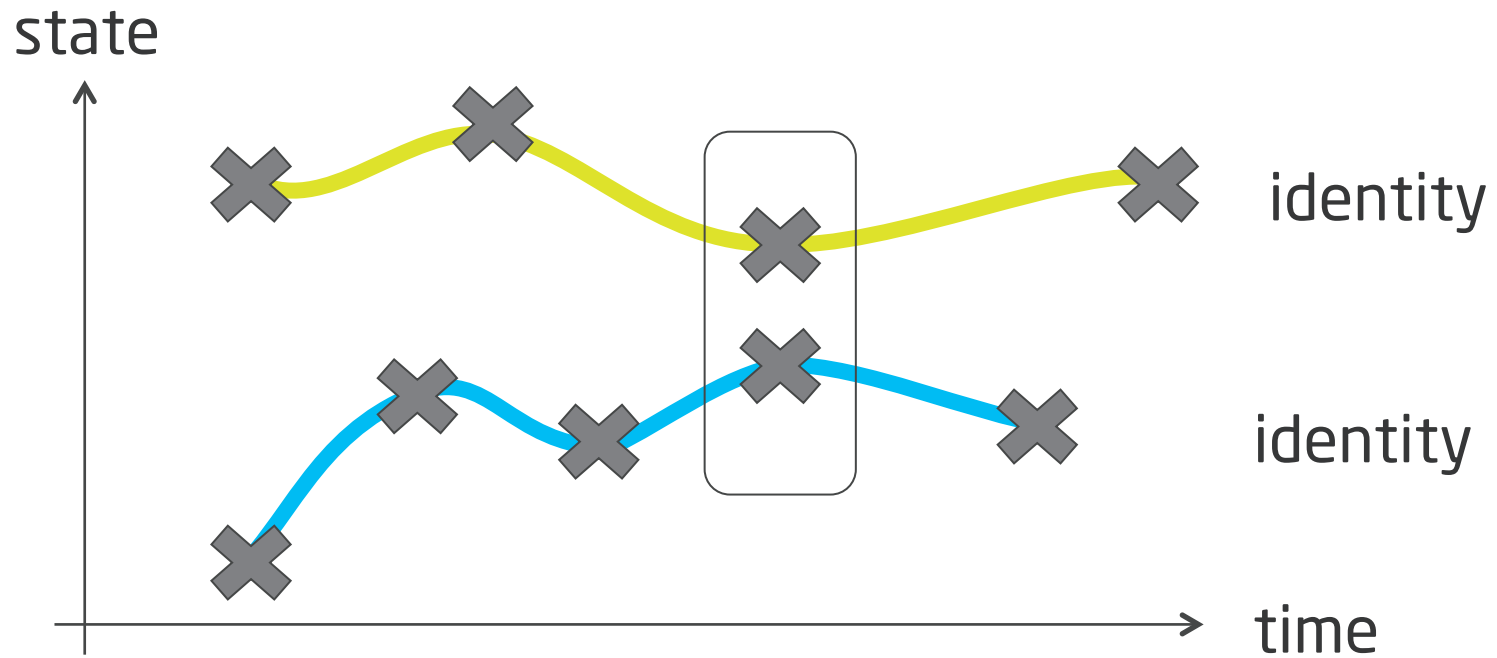
For full details see: Fast and Space Efficient Trie Searches, Bagwell [2000]

CONCURRENCY WITH SOFTWARE TRANSACTIONAL MEMORY

Concurrency Strategies

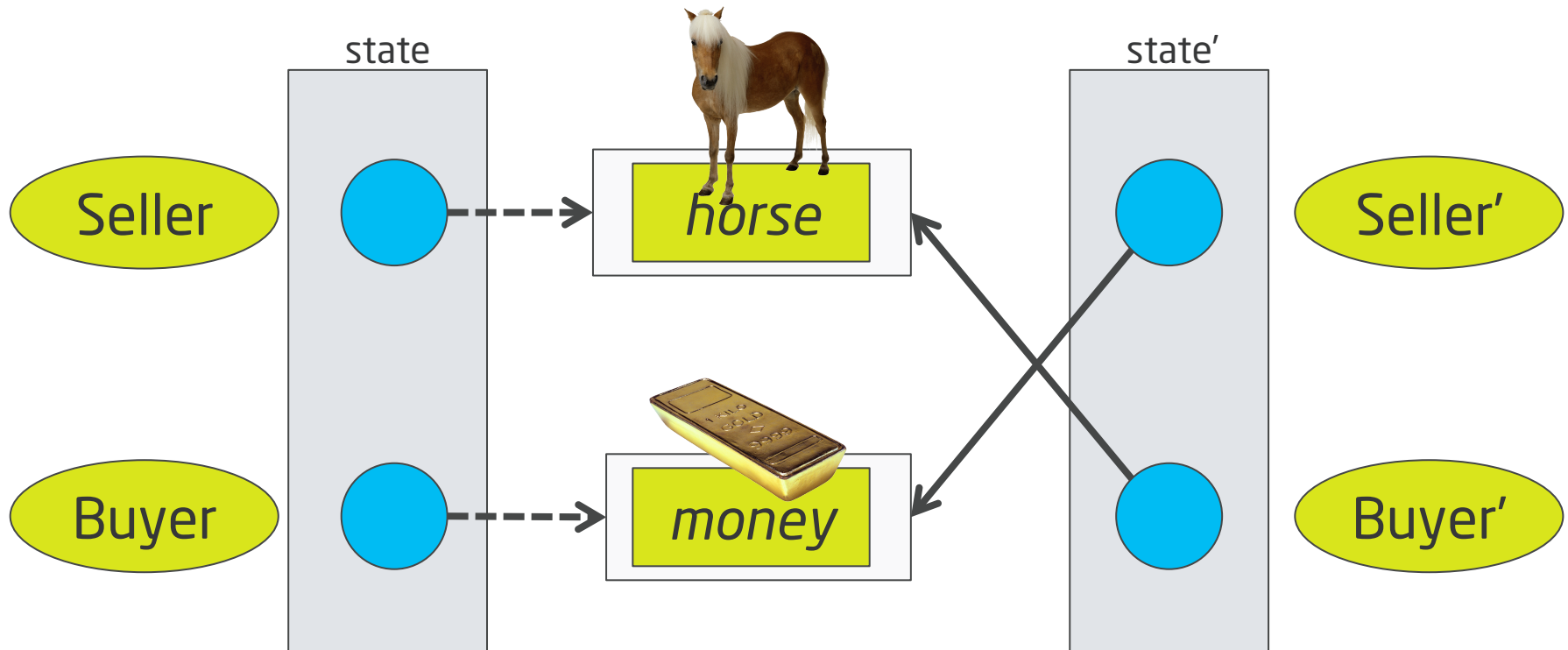


Clojure Concurrency



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

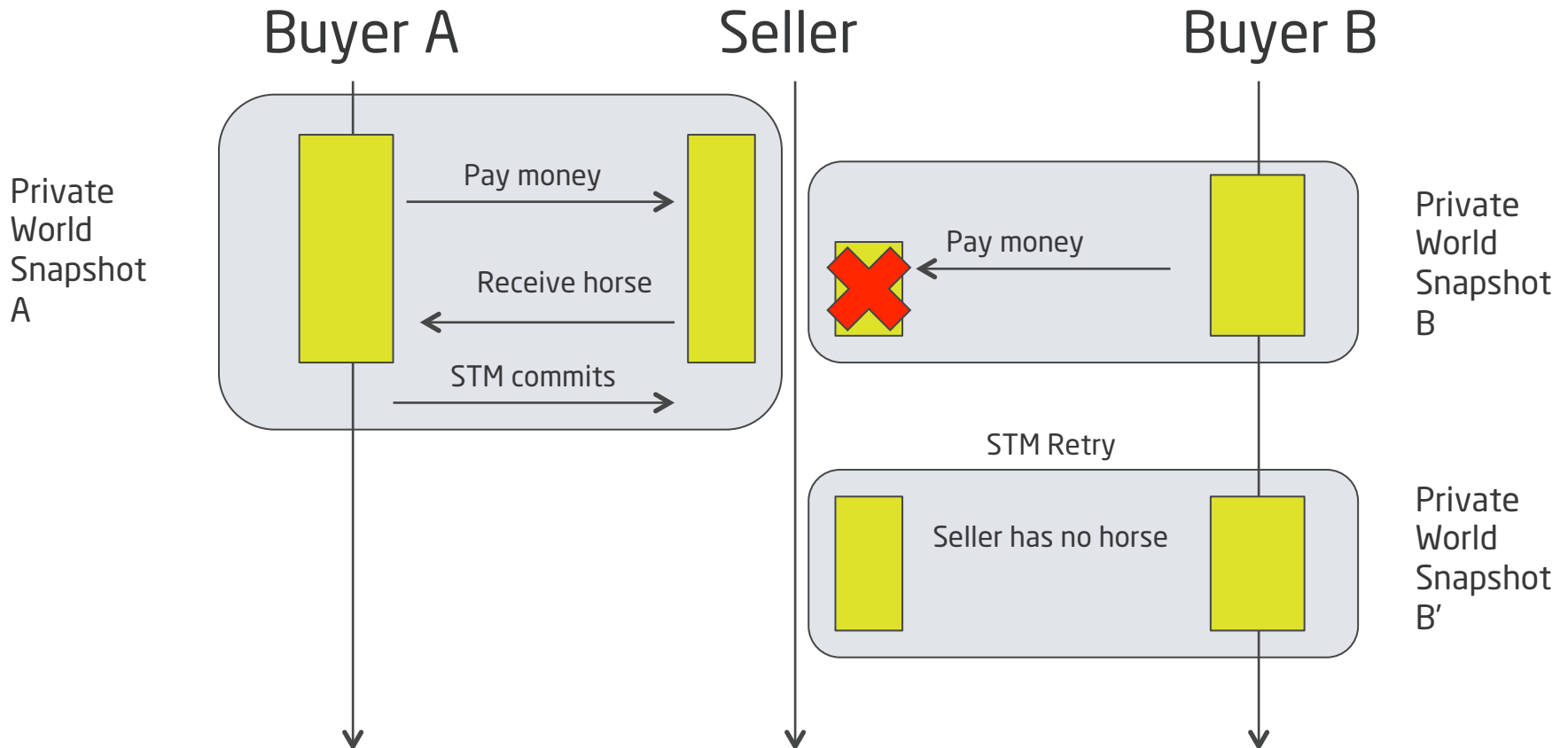
Software Transactional Memory



Multi-Version
Concurrency Control

Atomic
Consistent
Isolated
Durable

Software Transactional Memory Conflict Resolution



STM Example

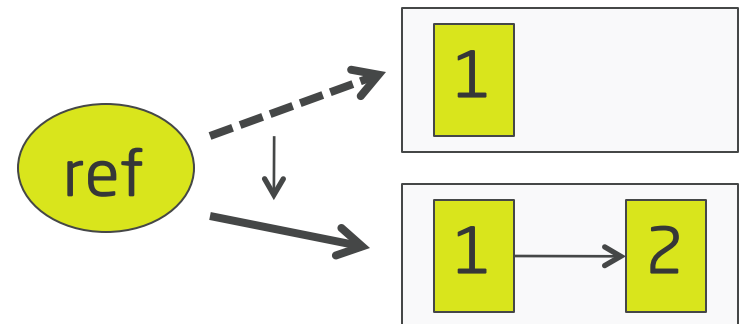
```
(defn post [account amount msg]
  (conj account {:amount amount, :msg msg}))
```

```
(defn transfer [from to amount msg]
  (dosync
    (alter from post (- amount) msg)
    (alter to post amount msg)))
```

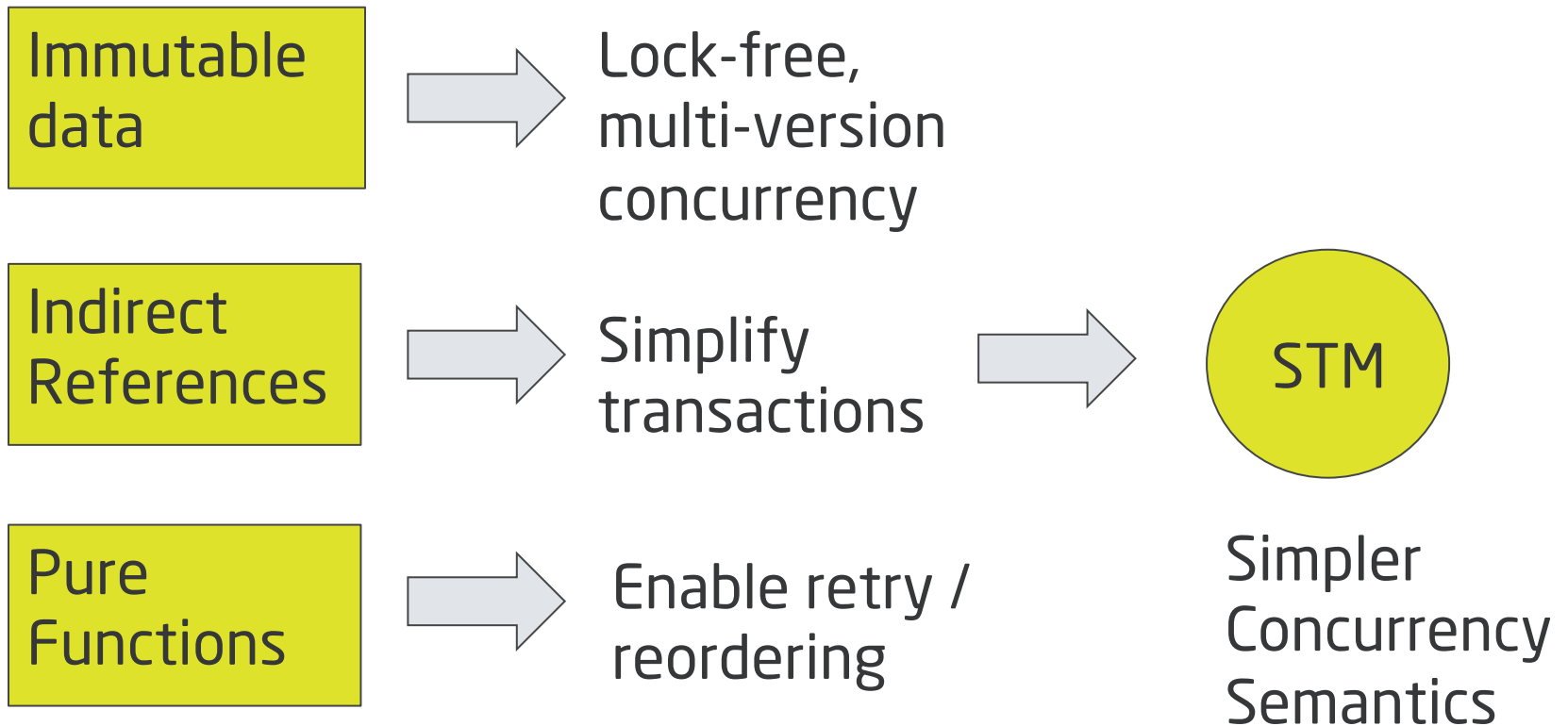
```
(defn balance [account]
  (reduce + 0 (map :amount account)))
```

```
(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)))))
```

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



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

```
(defrecord Conference [name year])  
  
(def ndc (Conference. "NDC" 2011))  
(def cc (Conference. "Clojure Conj" 2011))  
(def confs [ndc cc])
```

```
;; Records works with common functions  
(filter #(= 2011 (:year %)) confs)  
(assoc ndc :rating :great)
```

```
;; Their fields have map semantics  
(:year ndc)
```

```
;; A record is also a map of its properties  
(seq ndc)  
(doseq [[prop value] ndc]  
  (println prop "->" value))
```

file: islands.clj

**HOW WOULD YOU DO OBJECT
DIFF AND PATCH IN C#?**

Common Abstractions: diff

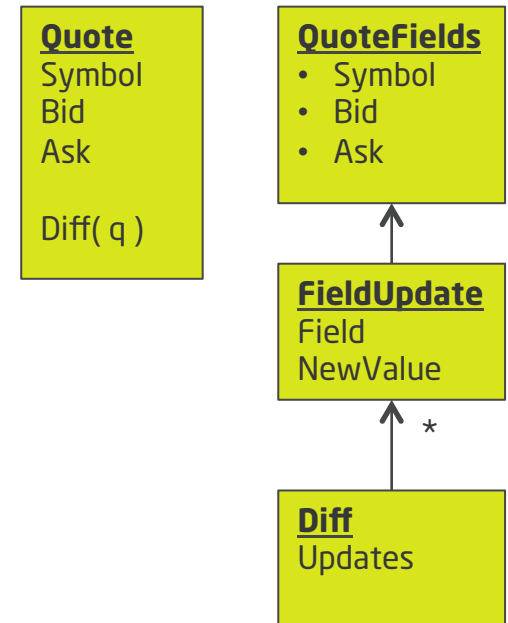
```
Diff CalculateDiff(Quote prev) {
    var result = new Diff();
    if (!(this.Bid.Equals(prev.Bid)))
        result.AddFieldUpdate(QuoteField.Bid, this.Bid);
    // repeat for the other fields ...
    return result
}
```

```
previous = Quote(EURUSD, Bid:=1.40, Ask:=1.41)
latest   = Quote(EURUSD, Bid:=1.45, Ask:=1.46)
```

```
diff = latest.ChangesSince(previous)
```

⇒ Diff with Updates:

```
(Field QuoteField.Bid, NewValue 1.405)
(Field QuoteField.Ask, NewValue 1.415)
```



Common Abstractions: diff

```
(defn diff [old new]
  (let [changed (filter (fn [k]
                        (not= (get old k) (get new k)))
                        (keys new))]
    (select-keys new changed)))
```

```
(defn patch [old df]
  (merge old df))
```

```
(diff {:bid 1.45, :ask 1.46, :symbol :eurusd}
      {:bid 1.44, :ask 1.47, :symbol :eurusd})
=> {:bid 1.44, :ask 1.47}
```

Code to Common Abstractions **A|T|I|V|E|**

Core Abstractions

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

Data Structures

<code>{ :key value }</code>	map
<code>[a b c]</code>	vector
<code>(1 2 3)</code>	list
<code>#{ :a :b :c }</code>	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) );
```

map

Produces a new sequence of the same length by applying a function to each member

```
(map f [1 2 3])  
⇒ ((f 1) (f 2) (f 3))
```

SELECT f(x) FROM xs

reduce

Reduce a sequence to a single element by combining the elements one by one

`(reduce + [1 2 3])`

$\Rightarrow 6$

`(reduce + [1 2 3])`

is `(+ (+ 1 2) 3)`

is `(+ 3 3)`

$\Rightarrow 6$

`SELECT SUM(x) FROM xs`

filter / remove

Select the matches or the non-matches from a seq

(filter even? [1 2 3 4])

⇒ (2 4)

(remove even? [1 2 3 4])

⇒ (1 3)

SELECT x FROM xs WHERE ...

SPECIALIZING THE IMPLEMENTATION LANGUAGE

How would you add an unless keyword to C#?

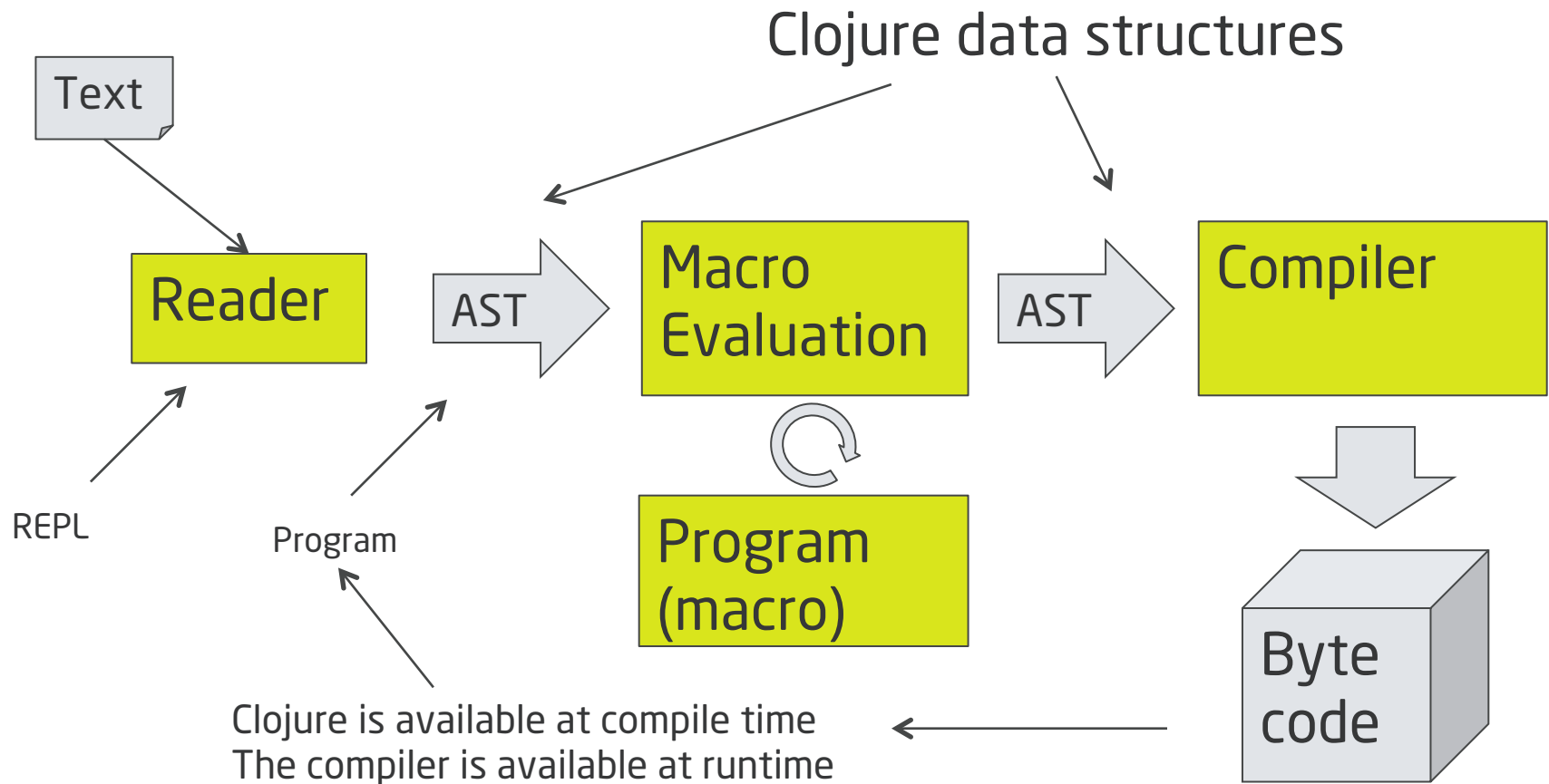
```
public WeakSetPerson(Person p)
{
    this.person = p unless (p == null);
}
```

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)))
```

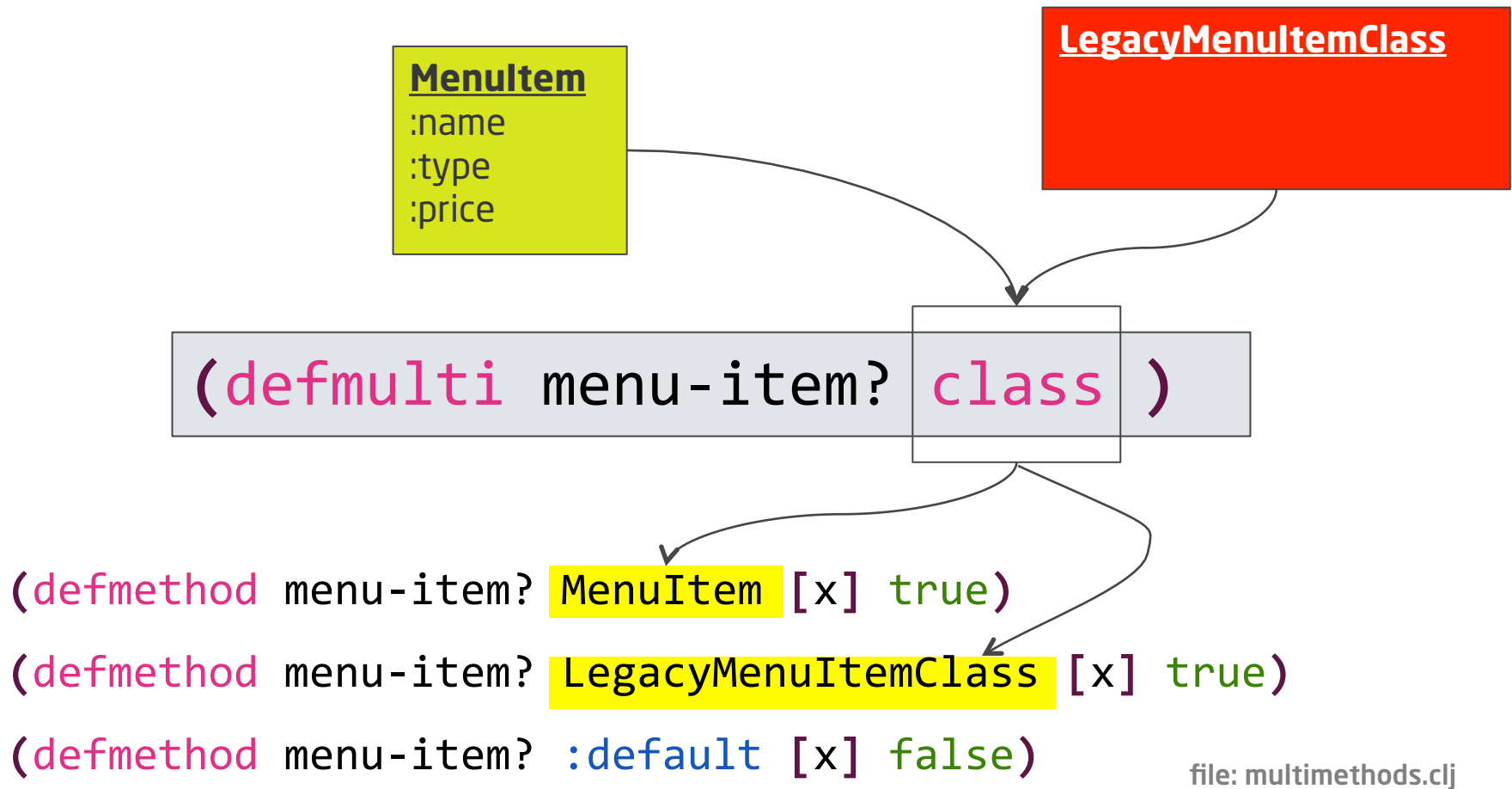
```
(macroexpand-1 '(unless (neg? x)  
                        (println "x is non-neg")))
```

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

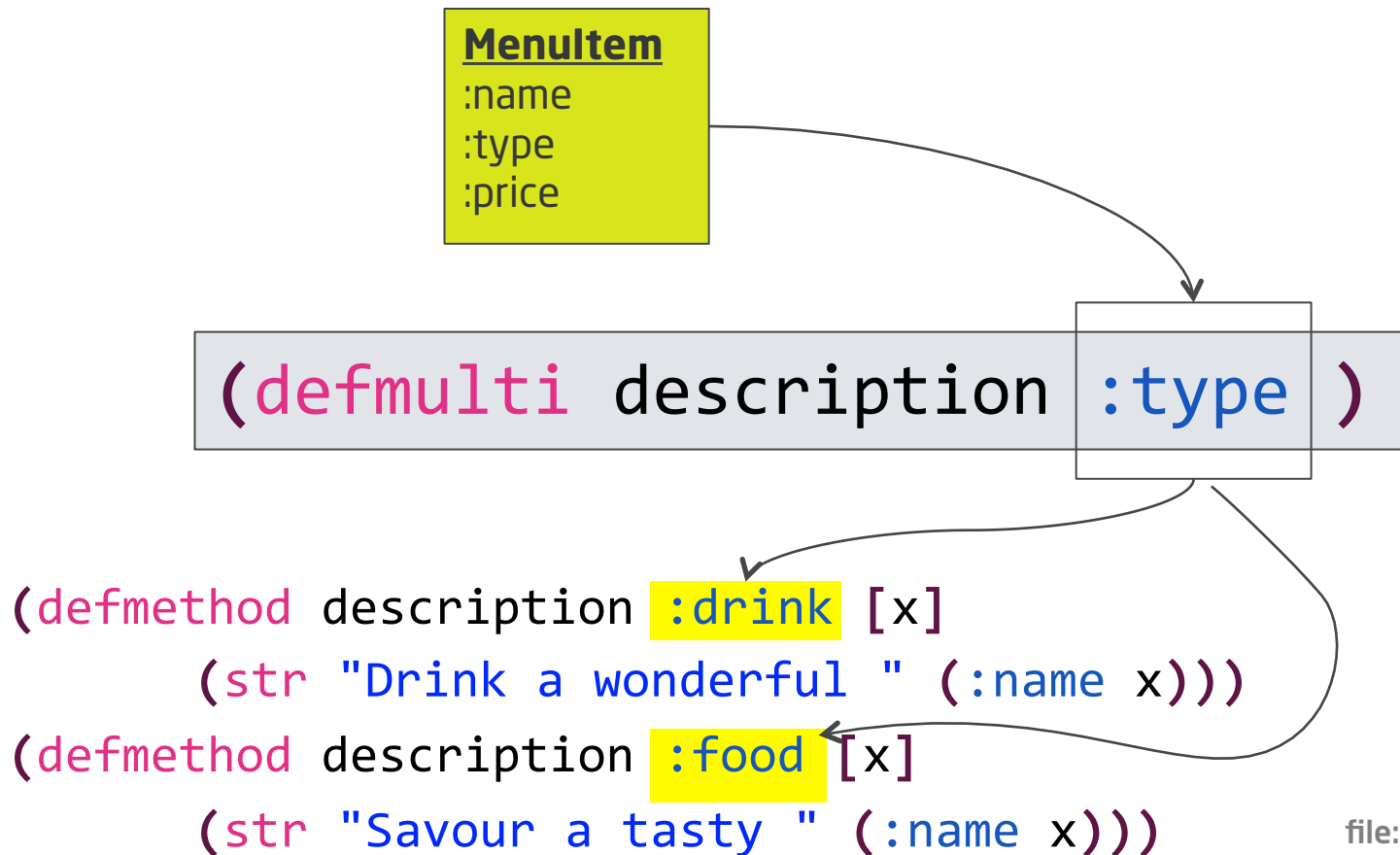
* Actually, this is the Clojure when-not macro

BETTER POLYMORPHISM

Open/Closed Legacy Code



Beyond Static Dispatch



CONCLUSIONS

Reducing 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"
- IntelliJ "La Clojure"
- Visual Studio "vsClojure"

Tools

- Cake (build, test +)
- Leiningen (ditto)
- Midje (testing)
- www.clojure.org

Thank you

Download the slides and examples here:

<https://github.com/mjul/top-10-clojure-ndc-2011>

Martin Jul

martin@mjul.com

Twitter: @mjul

Source

<https://github.com/mjul>

<https://github.com/ative>

Work and Blog

mj@ative.dk

<http://www.ative.dk>

[http://community.ative.dk/
blogs/](http://community.ative.dk/blogs/)