Functional Design Patterns

Aino Vonge Corry, PhD
Trifork A/S
Qcon London 2010

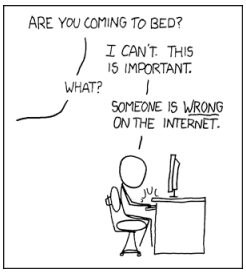
If you only remember one thing.

- Let it be this:
- Patterns are a useful tool of communication

Agenda

- Rumours
- Some stuff
- Some theory
- Some stuff
- Questions

Rumours



XKCD

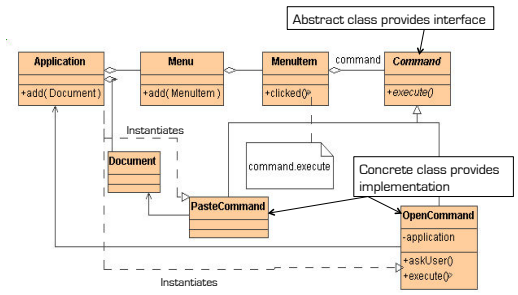
Command

- In a toolkit, you must include as much functionality as you can in order to make it useful.
- A user interface toolkit include objects like buttons and menus that carry out a request in response to user input.
- As toolkit designers we have no way of knowing the receiver of the request or the operations that will carry it out.

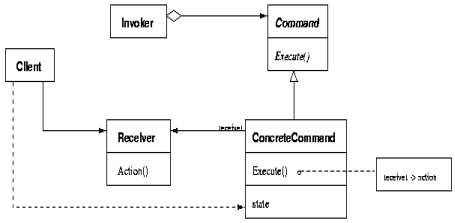
Solution

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations

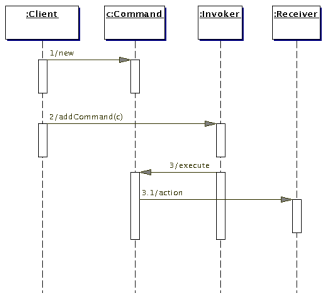
Static Solution



Static Structure



Dynamic solution



Consequences

- Command decouples the object that invokes the operation from the one that knows how to perform it.
- Commands are first-class objects. They can be manipulated and extended like any other object.
- It's easy to add new Commands, because you don't have to change existing classes.

Command in Erlang

```
start(D) ->
   spawn(doc, doc_controller, [D]).
    doc_controller(D) ->
       receive
          {get, Sender} ->
          Sender ! D,
             doc_controller(D);
          {transform, Fun} ->
             doc_controller(Fun(D))
end.
DC = doc:start("abc"),
Menu = menu:start(),
Menu ! {install_item, "Backwards",
        fun() -> DC ! {transform, fun lists:reverse/1} end},
```


GoF in dynamic languages

Accordig to Norvig 16 out of 23 GoF design patterns are invisible or made simpler by dynamic languages

First-class types: Abstract-factory, Factory-method, State, Proxy, Flyweight, Chain-of-responsibility

First-class functions: Command, Strategy, Template-method, Visitor

Macros: Interpreter, Iterator

Method combination: Mediator, Observer

Multimethods: Builder

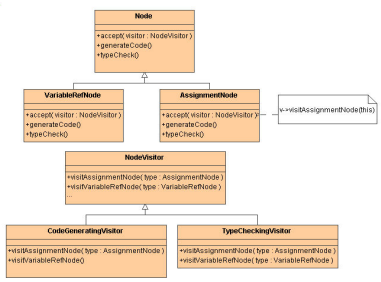
Modules: Facade

But we still have at least 7 left

- Composite, Singleton, Prototype, Adapter, Decorator, Memento

Problem

Visitor



Consequences

- Visitor makes it easy to extend the functionality.
- Collects related functions and separates unrelated.
- Makes it hard to extend the element hierarchy.

Visitor in FP

- Implementing Visitor in FP proves to be easy, because the wanted functionality of Visitor is to be function-oriented and take the functionality out of the abstract data types
- You can almost say it mimics functional programming

Visitor in FP

```
fun codegen (variableRef v) = (* generate variable v *)
  | codegen (assignment a) = (* generate assignment a *)

fun typecheck (variableRef v) = (* check type of v *)
  | typecheck (assignment a) = (* check types in a *)
```

Visitor in OO

```
Class Node {
Public:
   virtual void accept(Visitor&) = 0 ; }
}

Class NodeVisitor {
public:
   virtual void visitAssignmentNode(AssignmentNode*);
   virtual void VisitVariableRef(VariableRef*);
};
```


Visitor Class in OO

```
class AssignmentNode : public Node {
public:
   virtual void accept(Visitor& v) { v.VisitAssignmentNode(this); }
}

class CodeGeneratingVisitor : public NodeVisitor {
public:
   virtual void visitAssignmentNode(AssignmentNode*);
   virtual void VisitVariableRef(VariableRef*);
Private:
   Code _generated;
};

Void CodeGeneratingVisitor::visitAssignmentNode (AssignmentNode* a) {
   _generated = _generated ++ (* generate variable v *) }
```


Roundabout: Patterns for recursive programs

Eugene Wallingford

- **Structural Recursion** - use recursion to traverse inductively defined data structures (e.g. list or tree)
- **Interface Procedure** - use a helper method to adapt a function call if you need an extra argument
- **Mutual Recursion** - use multiple recursive functions to traverse inductively defined data structures where elements in the data structure are also inductively defined
- **Accumulator Variable** - use an additional parameter to carry calculations through recursion
- **Syntax Procedure** - extract methods that mean something rather than accessing raw members of data
- **Local Procedure** - use anonymous methods to hide details from the outside world
- **Program Derivation** - inline code if you want more performance

Structural Recursion

Consider the procedure replace, which operates on s-lists, with type definition:

```
<s-list> ::= ()| (<symbol-expression> . <s-list>)
<symbol-expression> ::= <symbol> | <s-list>

Gives:
>(replace a b ((ab)(((bgr)(fr))c(de))b))
                ((aa)(((agr)(fr))c(de))a))
```


Structural Recursion

```
(define replace
  (lambda (new old slist)
   (cond ((null? slist) '())
         ((symbol? (car slist))
           (if (eq? (car slist) old)
             (cons new (replace new old (cdr slist)))
             (cons (car slist) (replace new old (cdr slist)))))
         (#t (cons (replace new old (car slist))
                   (replace new old (cdr slist)))))))
```


Consequences

The data definition has two components, one for s-lists and one for symbol expressions.

These components are mutually inductive, that is, defined in terms of one another.

You want to be faithful to the structure of your data, because it simplifies the individual pieces of code and makes later changes to data definitions easier to incorporate.

Mutual Recursion (1:2)

```
(define replace
  (lambda (new old slist)
    (if (null? slist)
        '()
        (cons (replace-symbol-expr new old (car slist))
              (replace new old (cdr slist))))))
```


Mutual Recursion (2:2)

```
(define replace-symbol-expr
  (lambda (new old sym-expr)
    (if (symbol? sym-expr)
      (if (eq? sym-expr old)
        new
        sym-expr)
      (replace new old sym-expr))))
```

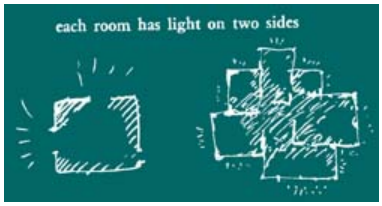

Consequences

- Truer to the data definition and does not repeat code
- But: can result in many immediate calls back to the calling procedure, thus inefficient

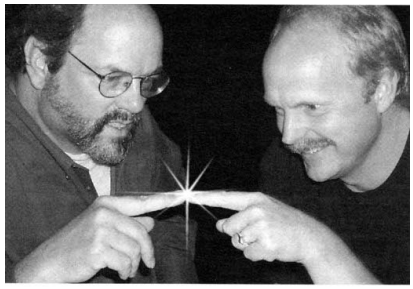
Program Derivation

```
(define replace
  (lambda (new old slist)
   (if (null? slist)
      '()
      (cons (if (symbol? (car slist))
              (if (eq? (car slist) old)
                 new
                 (car slist))
            (replace new old (car slist)))
            (replace new old (cdr slist))))))
```


The Pattern Concept



The first decade



Descriptions of Patterns

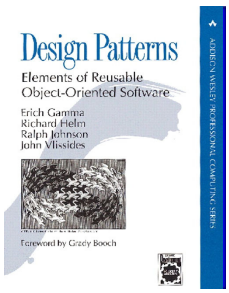
"Providing proven solutions to recurring design problems that arise in a given context"

Patterns document proven design experience — they exercise an "aggressive disregard for originality" (Brian Foote).

A Pattern Literary Form

- **Pattern Name**
- Intent/purpose
- Also known as/Aliases
- **Motivation/Context**
- Applicability/Forces
- **Solution**
- Structure
- Participants
- Collaborations
- **Consequences/Constraints**
- Implementation
- Sample Code
- Known Uses
- Related Patterns/Compare

A Pattern Example



How many times, have you thought:

'Boy, I sure wish there was an easier way to pick up women, like published API with code samples?'

What would you say if such documentation was not only available, but succinctly put into 22 design patterns and given formal descriptions just like the ones in your UML book?

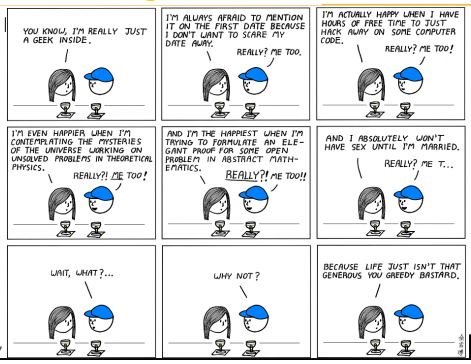
Surprise Statefulness

- Problem
  - You want to convince a target female that you are a package of extremely desirable resources and differentiate yourself from other dating service providers
- Forces
  - Women view men as somewhat self-centered
  - Women assign significant value to a man who takes the trouble to make her private data persistent
- Solution: Use optimistic persistence to implement explicit storage and retrieval of her private attributes

Surprise Statefulness

- Strategies
  - Standard text retrieval strategy (do you still use that wooden hula hoop ring?)
  - Object instantiation strategy (give her an old LP of of the first band she ever saw)
- Benefits and drawbacks
  - Dirty Reads
  - Mismatched data and client
  - Corresponding high return
  - Considerable investment up front
- Related Patterns
  - Interested Listener - listen

Abstrusegoose



Interested Listener

- Strategies
  - *askForDirectionsOrInformation, askHerAboutHerBook, askHerAdviceAboutSomething*
  - Implement **LookLikeYouAreListening**
- Benefits and drawbacks
  - Easier than thinkOfSomethingClever
  - More effective than seenYouHereBefore
  - Sometimes your data is stored in a friendZone cookie
- Related Patterns
  - Dating Savant

Enough theory

- Back to stuff

Memoization

- Problem
  - Ineffective code
- Solution
  - Save state to the extent needed
- Consequences
  - More effective code
  - You might need monads!!!

Fibonacci without Memoization
Haskell

```
slow_fib :: Int -> Integer
slow_fib 0 = 0
slow_fib 1 = 1
slow_fib n = slow_fib (n-2) + slow_fib (n-1)

Slow_fib(1000) =
slow_fib(998)+slow_fib(999)=
slow_fib(996)+slow_fib(997)+slow_fib(997)
   +slow_fib(998)
```


Fibonacci with Memoization
Haskell

```
memoized_fib :: Int -> Integer
memoized_fib =
  let fib 0 = 0
      fib 1 = 1
      fib n = memoized_fib (n-2) +memoized_fib (n-1)
  in  (map fib [0 ..] !!)
```

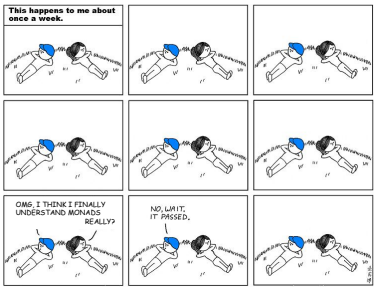
Fibonacci in Erlang

```
fibo(0) -> 0 ;
fibo(1) -> 1 ;
fibo(N) when N > 0 -> fibo(N-1) + fibo(N-2) .
```


Fibo with Memo in Erlang

```
fibo2(N) ->
{_,V} = fibo_memo(dict:new(), N), V.
fibo_memo(D,0) -> {D,0} ;
fibo_memo(D,1) -> {D,1} ;
fibo_memo(D,N) when N > 0 ->
  case dict:find(N,D) of
   {ok,Value} ->
      {D,Value};
   error ->
      {D0, F1} = fibo_memo(D,  N-1),
      {D1, F2} = fibo_memo(D0, N-2),
      Res = F1+F2,
      {dict:store(N,Res,D1), Res}
end.
```


Moment of Clarity (almost abstrusegoose)



Thank you for your time!

- Yes, some GoF patterns are not necessary in FP languages (as in some OO languages)
- Yes, it makes sense to talk about design patterns for FP
- Yes, they are as difficult to get an overview over as OO patterns