DCI: Practical Tips and Lessons for Nerds

Jim Coplien
Gertrud & Cope
Mørdrup, Denmark

With warm acknowledgment of
Steen Lehmann's Ruby code

Thanks to...

- Trygve Reenskaug
- Steen Lehmann
- Rickard Öberg
- Serge Beaumont
- Gertrud Bjørnvig
- Sadek Drobi

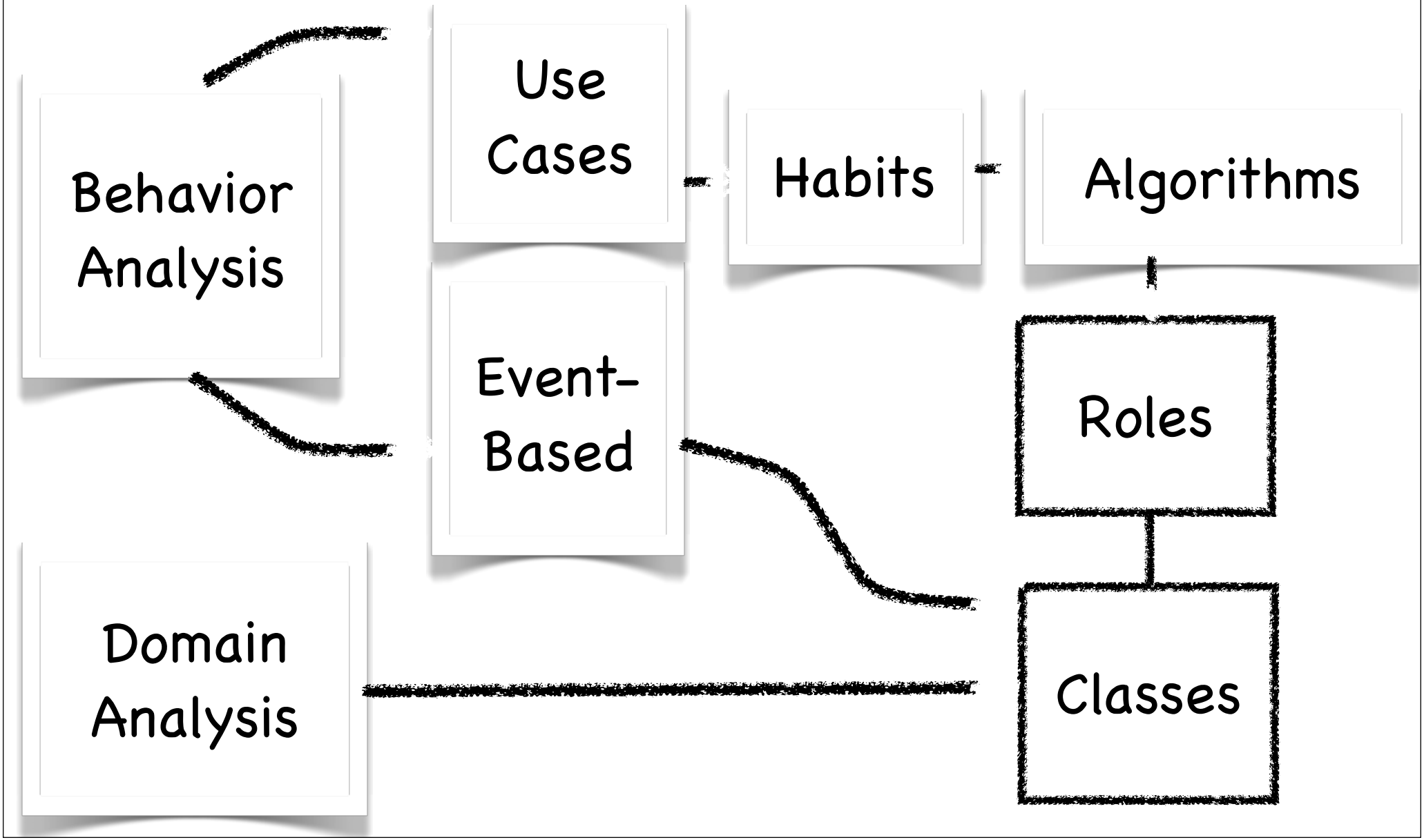
Languages

- Scala
- Python
- C#
- Javascript (Hi, Alan!)
- C++
- Java (Qi4J)
- Ruby
- PHP

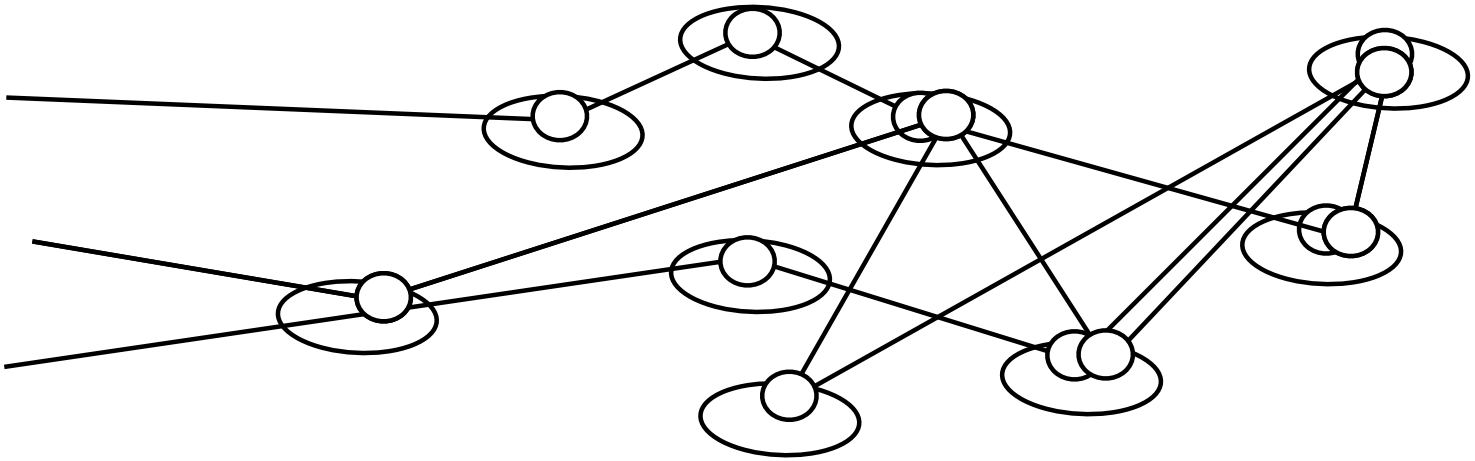
Two kinds of OO

Concern	Atomic Event	DCI Architecture
User Goal	Direct manipulation of a domain object	A sequence of tasks toward a goal
Requirements	State machine, custom formalism	Use Case
Technology	Good old OO	Multiparadigm design DCI
Design focus	Form of the data	Form of the algorithm
Scope	Single primary object or small static network	Multiple objects with dynamic associations
Interaction	Non-verb	Verb-noun
Example	Delete character Print balance	Spell check Money transfer

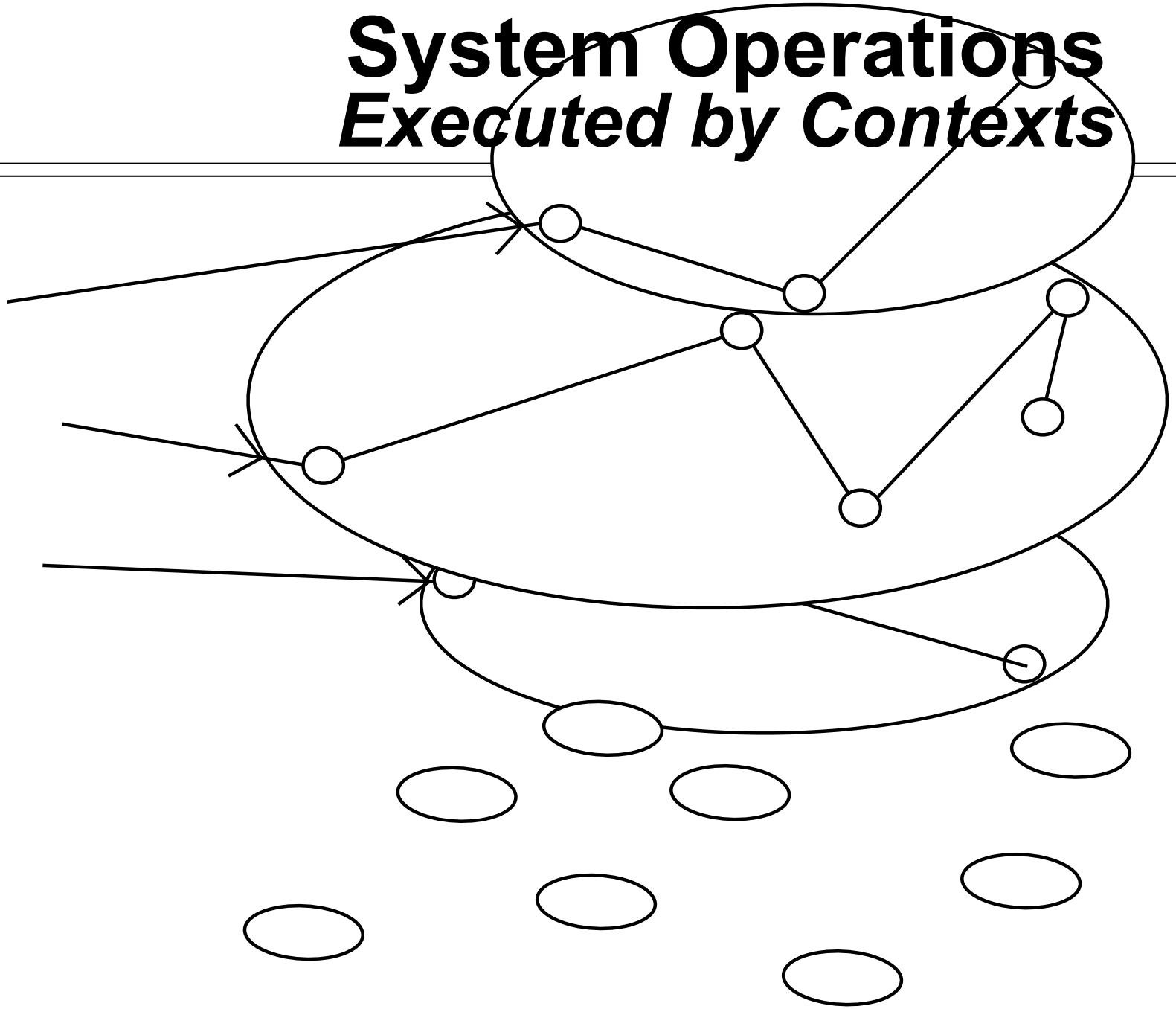
The process


Behavior Analysis
Use Cases
Habits
Algorithms
Event-Based
Roles
Domain Analysis
Classes

System Operations
Separation of Concern



System Operations
Executed by Contexts



A & D Concepts

- Use Cases: collections of scenarios between objects that achieve a goal in context
- Habits: Recurring Use Case fragments that lack a goal (e.g., "login")
- Event-Based: Atomic operations without a goal (e.g., "change color")
- Roles: Collections of related responsibilities
- Classes: Templates for dumb domain objects
- Algorithms: Steps that reflect sequencing constraints imposed by the implementation

Analysis: includes considered harmful

- A Use Case is best understood in terms of a business goal
- People have business goals; machine processes rarely do
- Therefore, a Use Case is best-suited to a human/computer interaction with a goal
- Breaking down Uses Cases causes some part of them to lose the goal
- Recurring fragments become habits

Domain Modeling: To Dumb Code

```
class SavingsAccount < Account

  # We will associate SavingsAccount with
  # TransferMoneySink at run time as needed

  def initialize(accountID, initialBalance)
    super(accountID, initialBalance)
  end
  private :initialize

  def availableBalance; @balance; end
  def decreaseBalance(amount); @balance -= amount; end
  def increaseBalance(amount); @balance += amount; end
  def updateLog(message, time, amount)
   . . . .
  end
 end
```

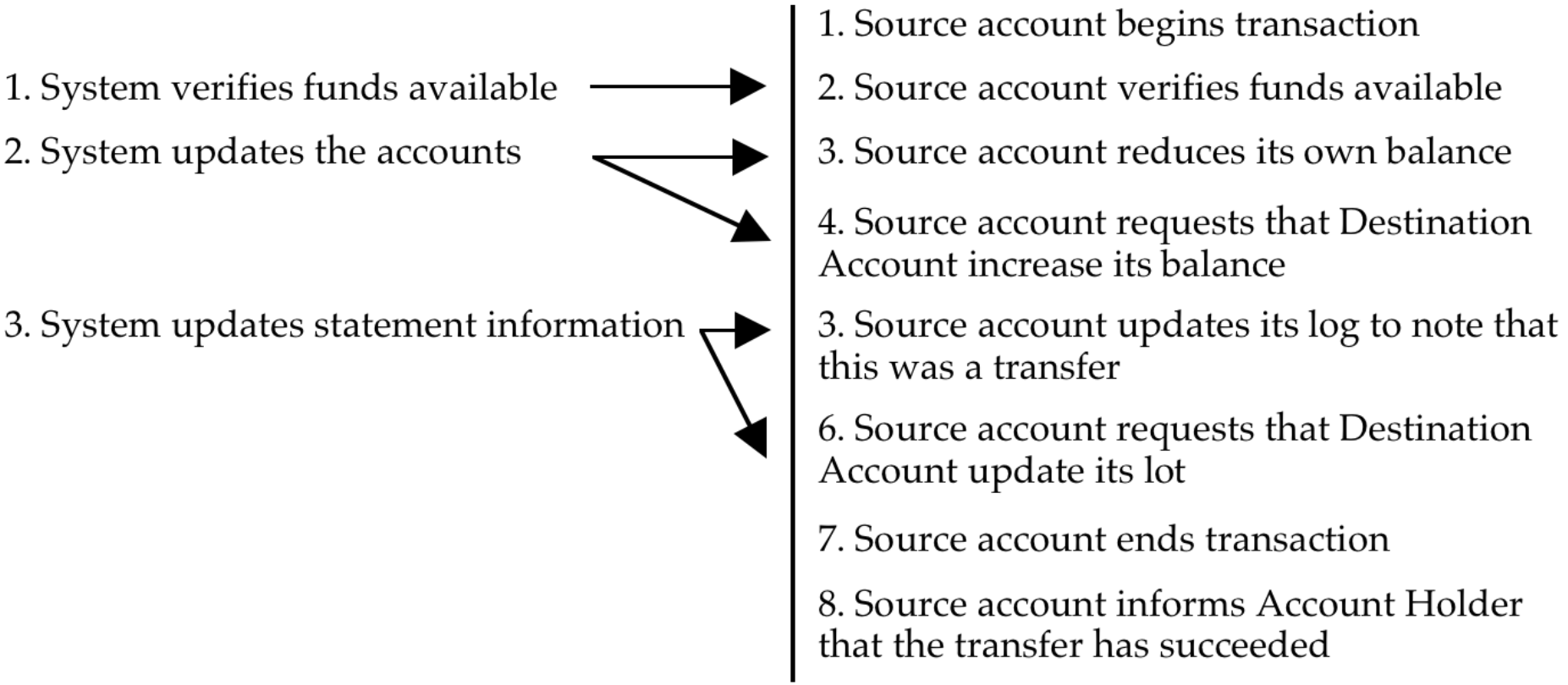
Basic DCI: Class Composition

```
class TransferMoneyContext

. . . .

def initialize(amt, sourceID, sinkID)
   @source_account = Account.find(sourceID)
   @source_account.extend TransferMoneySource

   @destination_account = Account.find(destID)
   @amount = amt
end

. . . .
```

In C++

```
class SavingsAccount: public
     TransferMoneySource<SavingsAccount>
{
public:
  void decreaseBalance(Currency amount) {
     . . . .
  }
   . . . .
};
```

Use Case to Algorithm



A Methodful Role

```
module TransferMoneySource
    include ContextAccessor

    # Role behaviors

    def transferTo
      raise "Insufficient funds" if balance < context.amount
      withdraw context.amount
      context.source_account.deposit context.amount
      self.updateLog "Transfer Out", Time.now, context.amount
      context.source_account.updateLog
            "Transfer In", Time.now, context.amount
      gui.displayScreen SUCCESS_DEPOSIT_SCREEN
      endTransaction
    end

end
```

No problem in C++

```
void transferTo(Currency amount) {
    // This code is reviewable and
    // meaningfully testable with stubs!
    beginTransaction();
    if (SELF->availableBalance() < amount) {
      endTransaction();
      throw InsufficientFunds();
    } else {
      SELF->decreaseBalance(amount);
      RECIPIENT->increaseBalance (amount);
      SELF->updateLog("Transfer Out", DateTime(),
                      amount);
      RECIPIENT->updateLog("Transfer In",
                      DateTime(), amount);
    }
    gui->displayScreen(SUCCESS_DEPOSIT_SCREEN);
    endTransaction();
```

"Calling" a context

```
def payBills
  # Assume that we can round
  # up the creditors
  creditors.each do |creditor|
    # transfer the funds here
 end
```

The Context class

```
class TransferMoneyContext
   attr_reader :source_account :destination_account,:amount
   . . . .
   def self.execute(amt, sourceID, sinkID)
    TransferMoneyContext.new(amt, sourceID,
          sinkID).execute
  end
  def initialize(amt, sourceID, sinkID)
    @source_account = Account.find(sourceID)
    @source_account.extend TransferMoneySource

    @destination_account = Account.find(destID)
    @amount = amt
  end
  def execute
    execute_in_context do
      source_account.transferTo
   end

end
```

"Calling" a context

```
def payBills
  # Assume that we can round
  # up the creditors
  creditors = context.creditors.dup
  creditors.each do |creditor|

  TransferMoneyContext.execute(
    creditor.amount_owed,
    account_id,
    creditor.account.account_id)
 end
```

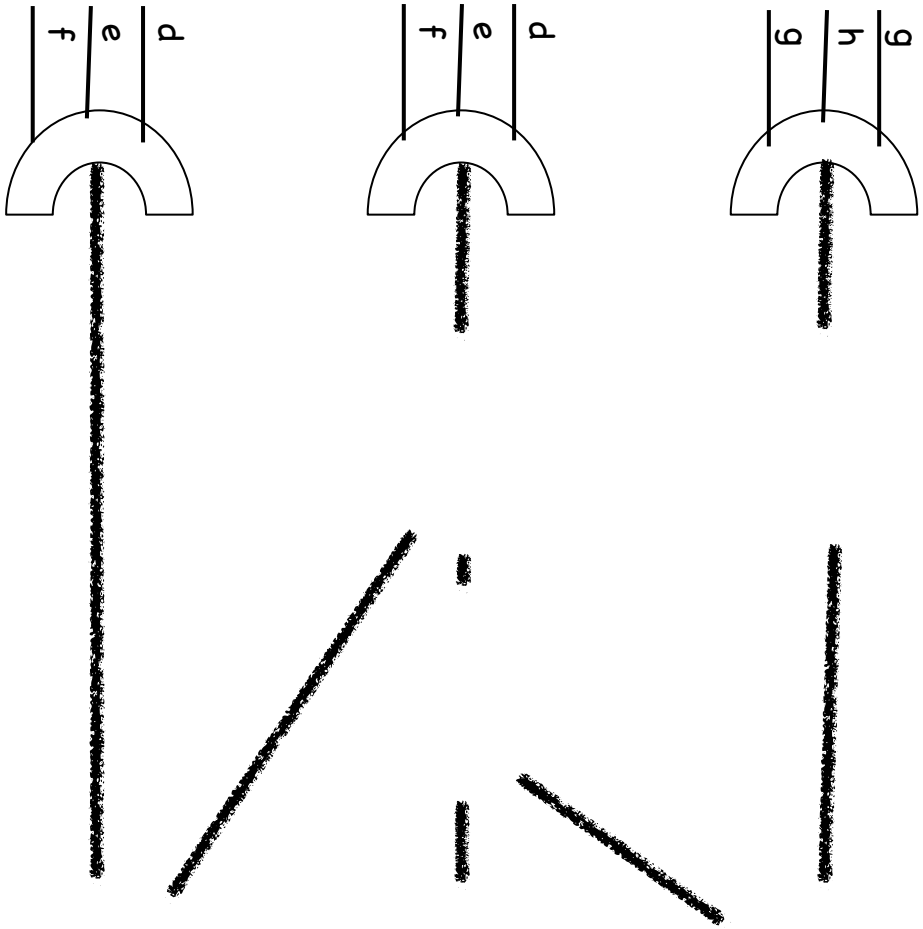
"Calling" a context

- A natural fit to habits
- Effectively the include relationship of Use Cases
- More than a subroutine call — includes role / object bindings

Accounts may not be Accounts

- Accounts are objects, right?
- Not really... the objects are transaction logs and audit trails
- Q: What is an Account? It IS part of the end user mental model, right?

A: Account is a Context



Dwellings

- It's clean code
- Architectural expressiveness only moderately better
- Maintainability radically better: a subtle effect
- I'm becoming more and more convinced of the need for a supporting environment

Questions?

- cope@gertrudandcope.com
- http://www.gertrudandcope.com-a.googlepages.com/thedciarchitecture